

The `lt3graph` package^{*†}

Michiel Helvensteijn
mhelvens+latex@gmail.com

January 5, 2017

Development of this package is organized at github.com/mhelvens/latex-lt3graph.
I am happy to receive feedback there!

1 Introduction

This package provides a data-structure for use in the $\text{\LaTeX}3$ programming environment. It allows you to represent a *directed graph*, which contains *vertices* (nodes), and *edges* (arrows) to connect them.¹ One such graph is defined below:

```
\ExplSyntaxOn
\graph_new:N \l_my_graph
\graph_put_vertex:Nn \l_my_graph {v}
\graph_put_vertex:Nn \l_my_graph {w}
\graph_put_vertex:Nn \l_my_graph {x}
\graph_put_vertex:Nn \l_my_graph {y}
\graph_put_vertex:Nn \l_my_graph {z}
\graph_put_edge:Nnn \l_my_graph {v} {w}
\graph_put_edge:Nnn \l_my_graph {w} {x}
\graph_put_edge:Nnn \l_my_graph {w} {y}
\graph_put_edge:Nnn \l_my_graph {w} {z}
\graph_put_edge:Nnn \l_my_graph {y} {z}
\graph_put_edge:Nnn \l_my_graph {z} {x}
\ExplSyntaxOff
```

Each vertex is identified by a *key*, which, to this library, is a string: a list of characters with category code 12 and spaces with category code 10. An edge is then declared between two vertices by referring to their keys.

We could then, for example, use TikZ to draw this graph:

^{*}This document corresponds to `lt3graph` v0.1.8, dated 2017/01/05.

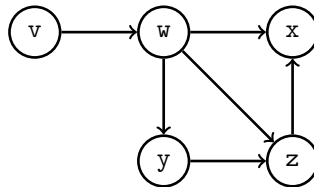
[†]The prefix `lt3` indicates that this package is a user-contributed `exp13` library, in contrast to packages prefixed with `l3`, which are officially supported by the $\text{\LaTeX}3$ team.

¹ Mathematically speaking, a directed graph is a tuple (V, E) with a set of vertices V and a set of edges $E \subseteq V \times V$ connecting those vertices.

```

\centering
\begin{tikzpicture}[every path/.style={line width=1pt,->}]
  \newcommand{\vrt}[1]{\node[#1]{\ttfamily\vphantom{Iy}#1}; }
  \matrix [nodes={circle,draw},
           row sep=1cm, column sep=1cm,
           execute at begin cell=\vrt] {
    v & w & x \\
    & y & z \\
  };
  \ExplSyntaxOn
  \graph_map_edges_inline:Nn \l_my_graph
  { \draw (#1) to (#2); }
  \ExplSyntaxOff
\end{tikzpicture}

```



Just to be clear, this library is *not about drawing* graphs. It does not, inherently, understand any TikZ. It is about *representing* graphs. This allows us to perform analysis on their structure. We could, for example, determine if there is a cycle in the graph:

```

\ExplSyntaxOn
  \graph_if_cyclic:NTF \l_my_graph {Yep} {Nope}
\ExplSyntaxOff

```

Nope

Indeed, there are no cycles in this graph. We can also list its vertices in topological order:

```

\ExplSyntaxOn
  \clist_new:N \LinearClist
  \graph_map_topological_order_inline:Nn \l_my_graph
  { \clist_put_right:Nn \LinearClist {\texttt{#1}} }
\ExplSyntaxOff
Visiting dependencies first: \(\ \LinearClist \)

```

Visiting dependencies first: v, w, y, z, x

There is a great deal more that can be done with graphs (some of which is even implemented in this library). A common use-case will be to attach data to vertices and/or edges. You could accomplish this with a property map from `l3prop`, but this library has already done that for you! Every vertex and every edge can store arbitrary token lists.²

In the next example we store the *degree* (the number of edges, both incoming and outgoing) of each vertex inside that vertex as data. We then query all vertices directly reachable from `w` and print their information in the output stream:

²This makes the mathematical representation of our graphs actually a 4-tuple (V, E, v, e) , where $v : V \rightarrow TL$ is a function that maps every vertex to a token list and $e : E \rightarrow TL$ is a function that maps every edge (i.e., pair of vertices) to a token list.

```

\ExplSyntaxOn
  \cs_generate_variant:Nn \graph_put_vertex:Nnn {Nnf}
  \graph_map_vertices_inline:Nn \l_my_graph {
    \graph_put_vertex:Nnf \l_my_graph {#1}
    { \graph_get_degree:Nn \l_my_graph {#1} }
  }
\ExplSyntaxOff

```

It's just an additional parameter on the `\graph_put_vertex` function. Edges can store data in the same way:

```

\ExplSyntaxOn
  \graph_map_edges_inline:Nn \l_my_graph {
    \graph_put_edge:Nnnn \l_my_graph {#1} {#2}
    { \int_eval:n{##1 * ##2} }
  }
\ExplSyntaxOff

```

The values `##1` and `##2` represent the data stored in, respectively, vertices `#1` and `#2`. This is a feature of `\graph_put_edge:Nnnn` added for your convenience.

We can show the resulting graph in a table, which is handy for debugging:

```

\ExplSyntaxOn \centering
  \graph_display_table:N \l_my_graph
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

The cells represent edges directly connecting two vertices. The (tr) cells don't have edges, but indicate that there is a sequence of edges connecting two vertices transitively.

Two vertices can have at most two arrows connecting them: one for each direction. If you want to represent a *multidigraph* (or *quiver*; I'm not making this up), you could consider storing a (pointer to a) list at each edge.

Finally, we demonstrate some transformation functions. The first generates the transitive closure of a graph:

```

\ExplSyntaxOn
  \graph_new:N \l_closed_graph
  \cs_new:Nn \__closure_combiner:nnn { #1,~#2,~(#3) }
  \graph_set_transitive_closure:NNNn
    \l_closed_graph \l_my_graph
    \__closure_combiner:nnn {--}
\ExplSyntaxOff

```

```

\ExplSyntaxOn \centering
\graph_display_table:N \l_my_graph
\(\ \ \rightsquigarrow\ \ \)
\graph_display_table:Nn \l_closed_graph
{ row_keys = false, vertex_vals = false }
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

 \rightsquigarrow

	v	w	x	y	z
v		4	4, 8, (-)	4, 8, (-)	4, 12, (-)
w			12, 6, (8)	8	8, 6, (12)
x					
y			6, 6, (-)		6
z			6		

There is a simpler version (`\graph_set_transitive_closure:NN`) that sets the values of the new edges to the empty token-list. The demonstrated version takes an expandable function to determine the new value, which has access to the values of the two edges being combined (as #1 and #2), as well as the value of the possibly already existing transitive edge (as #3). If there was no transitive edge there already, the value passed as #3 is the fourth argument of the transformation function; in this case --.

The second transformation function generates the transitive reduction:

```

\ExplSyntaxOn
\graph_new:N \l_reduced_graph
\graph_set_transitive_reduction:NN
\l_reduced_graph \l_my_graph
\ExplSyntaxOff

```

```

\ExplSyntaxOn \centering
\graph_display_table:N \l_my_graph
\(\ \ \rightsquigarrow\ \ \)
\graph_display_table:Nn \l_reduced_graph
{ row_keys = false, vertex_vals = false }
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

 \rightsquigarrow

	v	w	x	y	z
v		4	(tr)	(tr)	(tr)
w			(tr)	8	(tr)
x					
y			(tr)		6
z			6		

2 API Documentation

Sorry! There is no full API documentation yet. But in the meantime, much of the API is integrated in the examples of the previous section, and everything is documented (however sparsely) in the implementation below.

3 Implementation

We now show and explain the entire implementation from `lt3graph.sty`.

3.1 Package Info

```
1 \NeedsTeXFormat{LaTeX2e}
2 \RequirePackage{expl3}
3 \ProvidesExplPackage{lt3graph}{2017/01/05}{0.1.8}
4 {a LaTeX3 datastructure for representing directed graphs with data}
```

3.2 Required Packages

These are the packages we'll need:

```
5 \RequirePackage{l3keys2e}
6 \RequirePackage{xparse}
7 \RequirePackage{withargs}
```

3.3 Additions to L^AT_EX3 Fundamentals

These are three macros for working with ‘set literals’ in an expandable context. They use internal macros from `l3prop...`. Something I'm really not supposed to do.

```
8 \prg_new_conditional:Npnn \__graph_set_if_in:nn #1#2 { p }
9 {
10   \__prop_if_in:nwnn {#2} #1 \s_obj_end
11   \__prop_pair:wn #2 \s__prop { }
12   \q_recursion_tail
13   \__prg_break_point:
14 }
15
16 \cs_set_eq:NN \__graph_empty_set \s__prop
17
18 \cs_new:Nn \__graph_set_cons:nn {
19 #1 \__prop_pair:wn #2 \s__prop { }
20 }
```

3.4 Data Access

These functions generate the multi-part csnames under which all graph data is stored:

```
21 \cs_new:Nn \__graph_t1:n { g__graph_data (#1) _t1 }
22 \cs_new:Nn \__graph_t1:nn { g__graph_data (#1) (#2) _t1 }
23 \cs_new:Nn \__graph_t1:nnn { g__graph_data (#1) (#2) (#3) _t1 }
24 \cs_new:Nn \__graph_t1:nnnn { g__graph_data (#1) (#2) (#3) (#4) _t1 }
25 \cs_new:Nn \__graph_t1:nnnnn { g__graph_data (#1) (#2) (#3) (#4) (#5) _t1 }
```

The following functions generate multi-part keys to use in property maps:

```

26 \cs_new:Nn \__graph_key:n      { key (#1)                }
27 \cs_new:Nn \__graph_key:nn     { key (#1) (#2)           }
28 \cs_new:Nn \__graph_key:nnn   { key (#1) (#2) (#3)       }
29 \cs_new:Nn \__graph_key:nnnn  { key (#1) (#2) (#3) (#4)   }
30 \cs_new:Nn \__graph_key:nnnnn { key (#1) (#2) (#3) (#4) (#5) }

```

A quick way to iterate through property maps holding graph data:

```

31 \cs_new_protected:Nn \__graph_for_each_prop_datatype:n
32 { \seq_map_inline:Nn \g__graph_prop_data_types_seq {#1} }
33 \seq_new:N          \g__graph_prop_data_types_seq
34 \seq_set_from_clist:Nn \g__graph_prop_data_types_seq
35 {vertices, edge-values, edge-froms, edge-tos,
36  edge-triples, indegree, outdegree}

```

3.5 Storing data through pointers

The following function embodies a L^AT_EX3 design pattern for representing non-null pointers. This allows data to be 'protected' behind a macro redirection. Any number of expandable operations can be applied to the pointer indiscriminately without altering the data, even when using :x, :o or :f expansion. Expansion using :v dereferences the pointer and returns the data exactly as it was passed through #2. Expansion using :c returns a control sequence through which the data can be modified.

```

37 \cs_new_protected:Nn \__graph_ptr_new:Nn {
38   \withargs [\uniquecsname] {
39     \tl_set:Nn #1 {##1}
40     \tl_new:c   {##1}
41     \tl_set:cn {##1} {#2}
42   }
43 }
44 \cs_new_protected:Nn \__graph_ptr_gnew:Nn {
45   \withargs [\uniquecsname] {
46     \tl_gset:Nn #1 {##1}
47     \tl_new:c   {##1}
48     \tl_gset:cn {##1} {#2}
49   }
50 }

```

3.6 Creating and initializing graphs

Globally create a new graph:

```

51 \cs_new_protected:Nn \graph_new:N {
52   \graph_if_exist:NTF #1 {
53     % TODO: error
54   }{
55     \tl_new:N #1
56     \tl_set:Nf #1 { \tl_trim_spaces:f {\str_tail:n{#1}} }
57     \int_new:c {\__graph_tl:nnn{graph}{#1}{vertex-count}}
58     \__graph_for_each_prop_datatype:n
59     { \prop_new:c {\__graph_tl:nnn{graph}{#1}{##1}} }

```

```

60 }
61 }
62 \cs_generate_variant:Nn \tl_trim_spaces:n {f}

```

Remove all data from a graph:

```

63 \cs_new_protected:Nn \graph_clear:N
64 { \__graph_clear:Nn #1 { } }
65 \cs_new_protected:Nn \graph_gclear:N
66 { \__graph_clear:Nn #1 {g} }
67 \cs_new_protected:Nn \__graph_clear:Nn {
68   \__graph_for_each_prop_datatype:n
69   { \use:c{prop_#2clear:c} { \__graph_tl:nnn{graph}{#1}{##1} } }
70 }

```

Create a new graph if it doesn't already exist, then remove all data from it:

```

71 \cs_new_protected:Nn \graph_clear_new:N
72 { \__graph_clear_new:Nn #1 { } }
73 \cs_new_protected:Nn \graph_gclear_new:N
74 { \__graph_clear_new:Nn #1 {g} }
75 \cs_new_protected:Nn \__graph_clear_new:Nn {
76   \graph_if_exists:NF #1
77   { \graph_new:N #1 }
78   \use:c{graph_#2clear:N} #1
79 }

```

Set all data in graph #1 equal to that in graph #2:

```

80 \cs_new_protected:Nn \graph_set_eq:NN
81 { \__graph_set_eq:NNn #1 #2 { } }
82 \cs_new_protected:Nn \graph_gset_eq:NN
83 { \__graph_set_eq:NNn #1 #2 {g} }
84 \cs_new_protected:Nn \__graph_set_eq:NNn {
85   \use:c{graph_#3clear:N} #1
86   \__graph_for_each_prop_datatype:n
87   {
88     \use:c{prop_#3set_eq:cc}
89     { \__graph_tl:nnn{graph}{#1}{##1} }
90     { \__graph_tl:nnn{graph}{#2}{##1} }
91   }
92 }

```

An expandable test of whether a graph exists. It does not actually test whether the command sequence contains a graph and is essentially the same as `\cs_if_exist:N(TF)`:

```

93 \cs_set_eq:NN \graph_if_exist:Np \cs_if_exist:Np
94 \cs_set_eq:NN \graph_if_exist:NT \cs_if_exist:NT
95 \cs_set_eq:NN \graph_if_exist:NF \cs_if_exist:NF
96 \cs_set_eq:NN \graph_if_exist:NTF \cs_if_exist:NTF

```

3.7 Manipulating graphs

Put a new vertex inside a graph:

```

97 \cs_new_protected:Nn \graph_put_vertex:Nn
98   { \__graph_put_vertex:Nnnn #1 {#2} {} { } }
99 \cs_new_protected:Nn \graph_gput_vertex:Nn
100   { \__graph_put_vertex:Nnnn #1 {#2} {} {g} }
101 \cs_new_protected:Nn \graph_put_vertex:Nnn
102   { \__graph_put_vertex:Nnnn #1 {#2} {#3} { } }
103 \cs_new_protected:Nn \graph_gput_vertex:Nnn
104   { \__graph_put_vertex:Nnnn #1 {#2} {#3} {g} }
105 \cs_new_protected:Nn \__graph_put_vertex:Nnnn
106   {
107     %% create pointer to value
108     %
109     \use:c{__graph_ptr_#4new:Nn} \l__graph_vertex_data_tl {#3}
110
111     %% add the vertex
112     %
113     \use:c{prop_#4put:cnV} { \__graph_tl:nnn{graph}{#1}{vertices}}
114       {#2} \l__graph_vertex_data_tl
115
116     %% increment the vertex counter
117     %
118     \use:c{int_#4incr:c} { \__graph_tl:nnn{graph}{#1}{vertex-count}}
119
120     \graph_get_vertex:NnNT #1 {#2} \l_tmpa_tl {
121       %% initialize degree to 0
122       %
123       \use:c{prop_#4put:cnn} { \__graph_tl:nnn{graph}{#1}{indegree}} {#2} {0}
124       \use:c{prop_#4put:cnn} { \__graph_tl:nnn{graph}{#1}{outdegree}} {#2} {0}
125     }
126   }
127 \tl_new:N \l__graph_vertex_data_tl

```

Put a new edge inside a graph:

```

128 \cs_new_protected:Nn \graph_put_edge:Nnn
129   { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} { } }
130 \cs_new_protected:Nn \graph_gput_edge:Nnn
131   { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} {g} }
132 \cs_new_protected:Nn \graph_put_edge:Nnnn
133   { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} { } }
134 \cs_new_protected:Nn \graph_gput_edge:Nnnn
135   { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} {g} }
136 \cs_new_protected:Nn \__graph_put_edge:Nnnnn
137   {
138     \graph_get_vertex:NnNTF #1 {#2} \l__graph_from_value_tl {
139       \graph_get_vertex:NnNTF #1 {#3} \l__graph_to_value_tl {
140         \graph_get_edge:NnnNF #1 {#2} {#3} \l_tmpa_tl {
141           %% increment outgoing degree of vertex #2
142           %
143           \use:c{prop_#5put:cnf} { \__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
144             { \int_eval:n {

```



```

145         \prop_item:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} + 1
146     }}
147
148     %% increment incoming degree of vertex #3
149     %
150     \use:c{prop_#5put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
151     {\int_eval:n {
152         \prop_item:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} + 1
153     }}
154 }
155
156 %% actually add the edge
157 %
158 \withargs:VVn \l__graph_from_value_tl \l__graph_to_value_tl {
159     \use:c{prop_#5put:cox}
160     { \__graph_tl:nnn{graph}{#1}{edge-foms} }
161     { \__graph_key:nn{#2}{#3} }
162     { \tl_to_str:n{#2} }
163     \use:c{prop_#5put:cox}
164     { \__graph_tl:nnn{graph}{#1}{edge-tos} }
165     { \__graph_key:nn{#2}{#3} }
166     { \tl_to_str:n{#3} }
167     \use:c{__graph_ptr_#5new:Nn} \l__graph_edge_data_tl {#4}
168     \use:c{prop_#5put:coV}
169     { \__graph_tl:nnn{graph}{#1}{edge-values} }
170     { \__graph_key:nn{#2}{#3} }
171     \l__graph_edge_data_tl
172     \use:c{prop_#5put:cox}
173     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
174     { \__graph_key:nn{#2}{#3} }
175     { {\tl_to_str:n{#2}} }
176     { \tl_to_str:n{#3} }
177     { \l__graph_edge_data_tl }
178 }
179 ){
180     % TODO: Error ('to' vertex doesn't exist)
181 }
182 ){
183     % TODO: Error ('from' vertex doesn't exist)
184 }
185 }
186 \cs_generate_variant:Nn \prop_gput:Nnn {cox, coV, cnf}
187 \cs_generate_variant:Nn \prop_put:Nnn {cox, coV, cnf}
188 \cs_generate_variant:Nn \withargs:nnn {VVn}
189 \tl_new:N \l__graph_edge_data_tl
190 \tl_new:N \l__graph_from_value_tl
191 \tl_new:N \l__graph_to_value_tl

```

Remove a vertex from a graph, automatically removing any connected edges:

```

192 \cs_new_protected:Nn \graph_remove_vertex:Nn
193 { \__graph_remove_vertex:Nnn #1 {#2} { } }
194 \cs_new_protected:Nn \graph_gremove_vertex:Nn
195 { \__graph_remove_vertex:Nnn #1 {#2} {g} }

```

```

196 \cs_new_protected:Nn \__graph_remove_vertex:Nnn
197 {
198   \graph_get_vertex:NnNT #1 {#2} \l__graph_vertex_data_tl {
199     %%% remove outgoing edges
200     %
201     \graph_map_outgoing_edges_inline:Nnn #1 {#2}
202     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
203
204     %%% remove incoming edges
205     %
206     \graph_map_incoming_edges_inline:Nnn #1 {#2}
207     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
208
209     %%% remove the vertex
210     %
211     \use:c{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{vertices}} {#2}
212     \use:c{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{indegree}} {#2}
213     \use:c{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
214
215     %%% decrement the vertex counter
216     %
217     \use:c{int_#3decr:c} {\__graph_tl:nnn{graph}{#1}{vertex-count}}
218   }
219 }
220 \cs_generate_variant:Nn \prop_put:Nnn {cnV}
221 % \tl_new:N \l__graph_vertex_data_tl % reusing from other function

```

Remove an edge from the graph:

```

222 \cs_new_protected:Nn \graph_remove_edge:Nnn
223 { \__graph_remove_edge:Nnnn #1 {#2} {#3} { } }
224 \cs_new_protected:Nn \graph_gremove_edge:Nnn
225 { \__graph_remove_edge:Nnnn #1 {#2} {#3} {g} }
226 \cs_new_protected:Nn \__graph_remove_edge:Nnnn {
227   \graph_get_edge:NnnNT #1 {#2} {#3} \l__graph_edge_data_tl {
228     %%% decrement outdegree of vertex #2
229     %
230     \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
231     {\int_eval:n {
232       \prop_item:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} - 1
233     }}
234
235     %%% decrement indegree of vertex #3
236     %
237     \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
238     {\int_eval:n {
239       \prop_item:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} - 1
240     }}
241
242     %%% actually remove edge
243     %
244     \use:c{prop_#4remove:co}
245     { \__graph_tl:nnn{graph}{#1}{edge-frogs} }
246     { \__graph_key:nn{#2}{#3} }

```

```

247 \use:c{prop_#4remove:co}
248   { \_graph_tl:nnn{graph}{#1}{edge-tos}   }
249   { \_graph_key:nn{#2}{#3}               }
250 \use:c{prop_#4remove:co}
251   { \_graph_tl:nnn{graph}{#1}{edge-values} }
252   { \_graph_key:nn{#2}{#3}               }
253 \use:c{prop_#4remove:co}
254   { \_graph_tl:nnn{graph}{#1}{edge-triples} }
255   { \_graph_key:nn{#2}{#3}               }
256 }
257 }
258 \cs_generate_variant:Nn \prop_remove:Nn {co}
259 \cs_generate_variant:Nn \prop_gremove:Nn {co}
260 \cs_generate_variant:Nn \prop_put:Nnn {cnf}
261 \cs_generate_variant:Nn \prop_gput:Nnn {cnf}
262 %\tl_new:N \l_graph_edge_data_tl % reusing from other function

```

Add all edges from graph #2 to graph #1, but only between nodes already present in #1:

```

263 \cs_new_protected:Nn \graph_put_edges_from:NN
264   { \_graph_gput_edges_from:NNn #1 #2 { } }
265 \cs_new_protected:Nn \graph_gput_edges_from:NN
266   { \_graph_gput_edges_from:NNn #1 #2 {g} }
267 \cs_new_protected:Nn \_graph_gput_edges_from:NNn
268   {
269     \graph_map_edges_inline:Nn #2 {
270       \graph_if_vertex_exist:NnT #1 {##1} {
271         \graph_if_vertex_exist:NnT #1 {##2} {
272           \use:c{graph_#3put_edge:Nnn} #1 {##1} {##2} {##3}
273         }
274       }
275     }
276   }

```

3.8 Recovering values from graphs with branching

Test whether a vertex #2 exists. If so, its value is stored in #3 and T is left in the input stream. If it doesn't, F is left in the input stream.

```

277 \prg_new_protected_conditional:Nnn \graph_get_vertex:NnN
278   {T, F, TF}
279   {
280     \prop_get:cnNTF { \_graph_tl:nnn {graph} {#1} {vertices} } {#2} #3
281     { \tl_set:Nv #3 {#3} \prg_return_true: }
282     { \prg_return_false: }
283   }

```

Test whether an edge #2-#3 exists. If so, its value is stored in #4 and T is left in the input stream. If it doesn't, F is left in the input stream.

```

284 \prg_new_protected_conditional:Nnn \graph_get_edge:NnnN
285   {T, F, TF}
286   {
287     \prop_get:coNTF

```

```

288 { \_graph_tl:nnn{graph}{#1}{edge-values} }
289 { \_graph_key:nn{#2}{#3} }
290 #4
291 { \tl_set:Nv #4 {#4} \prg_return_true: }
292 { \prg_return_false: }
293 }

```

3.9 Graph Conditionals

An expandable test for the existence of a vertex:

```

294 \prg_new_conditional:Nnn \graph_if_vertex_exist:Nn
295 {p, T, F, TF}
296 {
297   \prop_if_in:cnTF
298   { \_graph_tl:nnn {graph} {#1} {vertices} }
299   { #2 }
300   { \prg_return_true: }
301   { \prg_return_false: }
302 }

```

An expandable test for the existence of an edge:

```

303 \prg_new_conditional:Nnn \graph_if_edge_exist:Nnn
304 {p, T, F, TF}
305 {
306   \prop_if_in:coTF
307   { \_graph_tl:nnn {graph} {#1} {edge-values} }
308   { \_graph_key:nn{#2}{#3} }
309   { \prg_return_true: }
310   { \prg_return_false: }
311 }

```

Test whether graph #1 contains a cycle reachable from vertex #2:

```

312 \cs_new:Npn \graph_if_vertex_can_reach_cycle_p:Nn #1#2
313 { \_graph_if_vertex_can_reach_cycle_p:Nnn #1 {#2} {\_graph_empty_set} }
314 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnTF #1#2
315 { \_graph_if_vertex_can_reach_cycle:NnnTF #1 {#2} {\_graph_empty_set} }
316 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnT #1#2
317 { \_graph_if_vertex_can_reach_cycle:NnnT #1 {#2} {\_graph_empty_set} }
318 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnF #1#2
319 { \_graph_if_vertex_can_reach_cycle:NnnF #1 {#2} {\_graph_empty_set} }
320
321 \prg_new_conditional:Nnn \_graph_if_vertex_can_reach_cycle:Nnn
322 {p, T, F, TF}
323 % #1: graph id
324 % #2: vertex id
325 % #3: visited vertices in 'prop literal' format (internal l3prop)
326 {
327   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
328   { \_graph_if_vertex_can_reach_cycle:Nnnnn #1 {#3} }
329   \prg_return_false:
330 }

```

```

331
332 \cs_new:Nn \__graph_if_vertex_can_reach_cycle:Nnnnn
333   % #1: graph id
334   % #2: visited vertices in 'prop literal' format (internal l3prop)
335   % #3: start vertex (not used)
336   % #4: current vertex
337   % #5: edge value (behind ptr, not used)
338   {
339     \bool_if:nT
340     {
341       \__graph_set_if_in_p:nn {#2} {#4} ||
342       \__graph_if_vertex_can_reach_cycle_p:Nno #1 {#4}
343         { \__graph_set_cons:nn {#2} {#4} }
344     }
345     { \prop_map_break:n {\use_i:nn \prg_return_true:} }
346   }
347 \cs_generate_variant:Nn \__graph_if_vertex_can_reach_cycle_p:Nnn {Nno}

```

Test whether graph #1 contains any cycles:

```

348 \prg_new_conditional:Nnn \graph_if_cyclic:N
349   {p, T, F, TF}
350   % #1: graph id
351   {
352     \graph_map_vertices_tokens:Nn #1
353     { \__graph_if_cyclic:Nnn #1 }
354     \prg_return_false:
355   }
356
357 \cs_new:Nn \__graph_if_cyclic:Nnn
358   % #1: graph id
359   % #2: vertex id
360   % #3: vertex value (not used)
361   {
362     \bool_if:nT
363     { \graph_if_vertex_can_reach_cycle_p:Nn #1 {#2} }
364     { \prop_map_break:n {\use_i:nn \prg_return_true:} }
365   }

```

Test whether graph #1 contains any cycles:

```

366 % \prg_new_protected_conditional:Nnn \graph_get_cycle:NN
367 %   {T, F, TF}
368 %   % #1: graph id
369 %   % #2: l3seq variable to put the cycle description in
370 %   {
371 %     \seq_clear:N #2
372 %     \__graph_get_cycle:NNTF #1 #2
373 %     {\prg_return_true: }
374 %     {\prg_return_false:}
375 %   }
376 %
377 % \prg_new_protected_conditional:Nnn \__graph_get_cycle:NN
378 %   {T, F, TF}

```

```

379 % % #1: graph id
380 % % #2: l3seq variable
381 % {
382 % \graph_map_successors_inline:Nnn #1 {} {
383 % \seq_if_in:NnTF #2 {##1} {
384 % % TODO
385 % }{
386 % % TODO
387 % }
388 % }
389 % }
390 %

```

Assume that graph #1 is acyclic and test whether a path exists from #2 to #3:

```

391 \prg_new_conditional:Nnn \graph_acyclic_if_path_exist:Nnn
392 {p, T, F, TF}
393 % #1: graph id
394 % #2: start vertex
395 % #3: end vertex
396 {
397 \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
398 { \graph_acyclic_if_path_exist:Nnnnn #1 {#3} }
399 \prg_return_false:
400 }
401
402 \cs_new:Nn \graph_acyclic_if_path_exist:Nnnnn
403 % #1: graph id
404 % #2: end vertex
405 % #3: start vertex (not used)
406 % #4: possible end vertex
407 % #5: edge value (behind ptr, do not use)
408 {
409 \bool_if:nT
410 {
411 \str_if_eq_p:nn {#4} {#2} ||
412 \graph_acyclic_if_path_exist_p:Nnn #1 {#4} {#2}
413 }
414 { \prop_map_break:n {\use_i:nn \prg_return_true:} }
415 }

```

3.10 Querying Information

Get the number of vertices in the graph:

```

416 \cs_new:Nn \graph_vertex_count:N {
417 \int_use:c {\graph_tl:nnn{graph}{#1}{vertex-count}}
418 }

```

Get the number of edges leading out of vertex #2:

```

419 \cs_new:Nn \graph_get_outdegree:Nn {

```

```

420 \prop_item:cn {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2}
421 }

```

Get the number of edges leading into vertex #2:

```

422 \cs_new:Nn \graph_get_indegree:Nn {
423   \prop_item:cn {\_graph_tl:nnn{graph}{#1}{indegree}} {#2}
424 }

```

Get the number of edges connected to vertex #2:

```

425 \cs_new:Nn \graph_get_degree:Nn {
426   \int_eval:n{ \graph_get_outdegree:Nn #1 {#2} +
427               \graph_get_indegree:Nn #1 {#2} }
428 }

```

3.11 Mapping Graphs

Applies the tokens #2 to all vertex name/value pairs in the graph. The tokens are supplied with two arguments as trailing brace groups.

```

429 \cs_new:Nn \graph_map_vertices_tokens:Nn {
430   \prop_map_tokens:cn
431     { \_graph_tl:nnn{graph}{#1}{vertices} }
432     { \_graph_map_vertices_tokens_aux:nnv {#2} }
433 }
434 \cs_new:Nn \_graph_map_vertices_tokens_aux:nnn
435   { #1 {#2} {#3} }
436 \cs_generate_variant:Nn \_graph_map_vertices_tokens_aux:nnn {nnv}

```

Applies the function #2 to all vertex name/value pairs in the graph. The function is supplied with two arguments as trailing brace groups.

```

437 \cs_new:Nn \graph_map_vertices_function:NN {
438   \prop_map_tokens:cn
439     { \_graph_tl:nnn{graph}{#1}{vertices} }
440     { \exp_args:Nnv #2 }
441 }

```

Applies the inline function #2 to all vertex name/value pairs in the graph. The inline function is supplied with two arguments: '#1' for the name, '#2' for the value.

```

442 \cs_new_protected:Nn \graph_map_vertices_inline:Nn {
443   \withargs (c) [\uniquecname] [#2] {
444     \cs_set:Npn ##1 #####2 {##2}
445     \graph_map_vertices_function:NN #1 #1
446   }
447 }

```

Applies the tokens #2 to all edge from/to/value triples in the graph. The tokens are supplied with three arguments as trailing brace groups.

```

448 \cs_new:Nn \graph_map_edges_tokens:Nn {
449   \prop_map_tokens:cn
450   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
451   { \__graph_map_edges_tokens_aux:nnn {#2} }
452 }
453 \cs_new:Nn \__graph_map_edges_tokens_aux:nnn
454   { \__graph_map_edges_tokens_aux:nnnv {#1} #3 }
455 \cs_new:Nn \__graph_map_edges_tokens_aux:nnnn
456   { #1 {#2} {#3} {#4} }
457 \cs_generate_variant:Nn \__graph_map_edges_tokens_aux:nnnn {nnnv}

```

Applies the function #2 to all edge from/to/value triples in the graph. The function is supplied with three arguments as trailing brace groups.

```

458 \cs_new:Nn \graph_map_edges_function:NN {
459   \prop_map_tokens:cn
460   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
461   { \__graph_map_edges_function_aux:Nnn #2 }
462 }
463 \cs_new:Nn \__graph_map_edges_function_aux:Nnn
464   { \__graph_map_edges_function_aux:Nnnv #1 #3 }
465 \cs_new:Nn \__graph_map_edges_function_aux:Nnnn
466   { #1 {#2} {#3} {#4} }
467 \cs_generate_variant:Nn \__graph_map_edges_function_aux:Nnnn {Nnnv}

```

Applies the tokens #2 to all edge from/to/value triples in the graph. The tokens are supplied with three arguments: ‘#1’ for the ‘from’ vertex, ‘#2’ for the ‘to’ vertex and ‘#3’ for the edge value.

```

468 \cs_new_protected:Nn \graph_map_edges_inline:Nn {
469   \withargs (c) [\uniquecsname] [#2] {
470     \cs_set:Npn ##1 #####2####3 {##2}
471     \graph_map_edges_function:NN #1 ##1
472   }
473 }

```

Applies the tokens #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The tokens are supplied with three arguments as trailing brace groups.

```

474 \cs_new:Nn \graph_map_incoming_edges_tokens:Nnn {
475   % #1: graph
476   % #2: base vertex
477   % #3: tokens to execute
478   \prop_map_tokens:cn
479   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
480   { \__graph_map_incoming_edges_tokens_aux:nnnn {#2} {#3} }
481 }
482 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnn
483   % #1: base vertex
484   % #2: tokens to execute
485   % #3: edge key
486   % #4: edge-triple {from}{to}{value}
487   { \__graph_map_incoming_edges_tokens_aux:nnnv {#1} {#2} #4 }

```



```

488 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn
489 % #1: base vertex
490 % #2: tokens to execute
491 % #3: edge 'from' vertex
492 % #4: edge 'to' vertex
493 % #5: edge value
494 { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
495 \cs_generate_variant:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn {nnnnv}

```

Applies the function #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The function is supplied with three arguments as trailing brace groups.

```

496 \cs_new:Nn \graph_map_incoming_edges_function:NnN {
497 % #1: graph
498 % #2: base vertex
499 % #3: function to execute
500 \prop_map_tokens:cn
501 { \__graph_tl:nnn{graph}{#1}{edge-triples} }
502 { \__graph_map_incoming_edges_function_aux:nNnn {#2} #3 }
503 }
504 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnn
505 % #1: base vertex
506 % #2: function to execute
507 % #3: edge key
508 % #4: edge-triple {from}{to}{value}
509 { \__graph_map_incoming_edges_function_aux:nNnnv {#1} #2 #4 }
510 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnnn
511 % #1: base vertex
512 % #2: function to execute
513 % #3: edge 'from' vertex
514 % #4: edge 'to' vertex
515 % #5: edge value
516 { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
517 \cs_generate_variant:Nn \__graph_map_incoming_edges_function_aux:nNnnn {nNnnv}

```

Applies the inline function #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The inline function is supplied with three arguments: ‘#1’ for the ‘from’ vertex, ‘#2’ is equal to the #2 supplied to this function and ‘#3’ contains the edge value.

```

518 \cs_new_protected:Nn \graph_map_incoming_edges_inline:Nnn {
519 % #1: graph
520 % #2: base vertex
521 % #3: body to execute
522 \withargs (c) [\uniquecsname] [#2] [#3] {
523 \cs_set:Npn ##1 #####2####3 {##3}
524 \graph_map_incoming_edges_function:NnN #1 {##2} ##1
525 }
526 }

```

Applies the tokens #3 to the from/to/value triples for the edges going ‘from’ vertex #2. The tokens are supplied with three arguments as trailing brace groups.

```

527 \cs_new:Nn \graph_map_outgoing_edges_tokens:Nnn {
528   % #1: graph
529   % #2: base vertex
530   % #3: tokens to execute
531   \prop_map_tokens:cn
532   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
533   { \__graph_map_outgoing_edges_tokens_aux:nnnn {#2} {#3} }
534 }
535 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnn
536   % #1: base vertex
537   % #2: tokens to execute
538   % #3: edge key (not used)
539   % #4: edge-triple {from}{to}{value}
540   { \__graph_map_outgoing_edges_tokens_aux:nnnnv {#1} {#2} #4 }
541 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn
542   % #1: base vertex
543   % #2: tokens to execute
544   % #3: edge 'from' vertex
545   % #4: edge 'to' vertex
546   % #5: edge value
547   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
548 \cs_generate_variant:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn {nNnnv}

```

Applies the function #3 to the from/to/value triples for the edges going 'from' vertex #2. The function is supplied with three arguments as trailing brace groups.

```

549 \cs_new:Nn \graph_map_outgoing_edges_function:NnN {
550   % #1: graph
551   % #2: base vertex
552   % #3: function to execute
553   \prop_map_tokens:cn
554   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
555   { \__graph_map_outgoing_edges_function_aux:nNnn {#2} #3 }
556 }
557 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnn
558   % #1: base vertex
559   % #2: function to execute
560   % #3: edge key
561   % #4: edge-triple {from}{to}{value}
562   { \__graph_map_outgoing_edges_function_aux:nNnnv {#1} #2 #4 }
563 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnnn
564   % #1: base vertex
565   % #2: function to execute
566   % #3: edge 'from' vertex
567   % #4: edge 'to' vertex
568   % #5: edge value
569   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
570 \cs_generate_variant:Nn \__graph_map_outgoing_edges_function_aux:nNnnn {nNnnv}

```

Applies the inline function #3 to the from/to/value triples for the edges going 'from' vertex #2. The inline function is supplied with three arguments: '#1' is equal to the #2 supplied to this function, '#2' contains the 'to' vertex and '#3' contains the edge value.

```

571 \cs_new_protected:Nn \graph_map_outgoing_edges_inline:Nnn {
572   % #1: graph
573   % #2: base vertex
574   % #3: body to execute
575   \withargs (c) [\uniquecsname] [#2] [#3] {
576     \cs_set:Npn ##1 #####2####3 {##3}
577     \graph_map_outgoing_edges_function:NnN #1 {##2} ##1
578   }
579 }

```

Applies the tokens #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The tokens are supplied with two arguments as trailing brace groups.

```

580 \cs_new:Nn \graph_map_successors_tokens:Nnn {
581   % #1: graph
582   % #2: base vertex
583   % #3: tokens to execute
584   \prop_map_tokens:cn
585   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
586   { \__graph_map_successors_tokens_aux:Nnnnn #1 {#2} {#3} }
587 }
588 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnn {
589   % #1: the graph
590   % #2: base vertex
591   % #3: tokens to execute
592   % #4: edge key (not used)
593   % #5: edge-triple {from}{to}{value}
594   \__graph_map_successors_tokens_aux:Nnnnnn #1 {#2} {#3} #5
595 }
596 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnnn {
597   % #1: the graph
598   % #2: base vertex
599   % #3: tokens to execute
600   % #4: edge 'from' vertex
601   % #5: edge 'to' vertex
602   % #6: ptr to edge value (not used)
603   \str_if_eq:nnT {#2} {#4} {
604     \__graph_map_successors_tokens_aux:nnv
605     {#3} {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
606   }
607 }
608 \cs_new:Nn \__graph_map_successors_tokens_aux:nnn {
609   % #1: tokens to execute
610   % #2: successor key
611   % #3: successor value
612   #1 {#2} {#3}
613 }
614 \cs_generate_variant:Nn \__graph_map_successors_tokens_aux:nnn {nnv}

```

Applies the function #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The function is supplied with two arguments as trailing brace groups.

```

615 \cs_new:Nn \graph_map_successors_function:NnN {
616   % #1: graph
617   % #2: base vertex
618   % #3: function to execute
619   \prop_map_tokens:cn
620   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
621   { \__graph_map_successors_function_aux:NnNnn #1 {#2} #3 }
622 }
623 \cs_new:Nn \__graph_map_successors_function_aux:NnNnn {
624   % #1: the graph
625   % #2: base vertex
626   % #3: function to execute
627   % #4: edge key (not used)
628   % #5: edge-triple {from}{to}{value}
629   \__graph_map_successors_function_aux:NnNnnn #1 {#2} #3 #5
630 }
631 \cs_new:Nn \__graph_map_successors_function_aux:NnNnnn {
632   % #1: the graph
633   % #2: base vertex
634   % #3: function to execute
635   % #4: edge 'from' vertex
636   % #5: edge 'to' vertex
637   % #6: ptr to edge value (not used)
638   \str_if_eq:nnT {#2} {#4} {
639     \__graph_map_successors_function_aux:Nnv
640     #3 {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
641   }
642 }
643 \cs_new:Nn \__graph_map_successors_function_aux:Nnn {
644   % #1: function to execute
645   % #2: successor key
646   % #3: successor value
647   #1 {#2} {#3}
648 }
649 \cs_generate_variant:Nn \__graph_map_successors_function_aux:Nnn {Nnv}

```

Applies the inline function #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The inline function is supplied with two arguments: ‘#1’ is the key, and ‘#2’ is the value of the successor vertex.

```

650 \cs_new_protected:Nn \graph_map_successors_inline:Nnn {
651   % #1: graph
652   % #2: base vertex
653   % #3: body to execute
654   \withargs (c) [\uniquecsname] [#2] [#3] {
655     \cs_set:Npn ##1 #####2####3 {##3}
656     \graph_map_successors_function:NnN #1 {##2} ##1
657   }
658 }

```

Applies the tokens #2 to all vertex name/value pairs in topological order. The tokens are supplied with two arguments as trailing brace groups. Assumes that the graph is acyclic (for now).

```

659 \cs_new_protected:Nn \graph_map_topological_order_tokens:Nn {
660
661   %% Fill \l__graph_source_vertices with source-nodes and count indegrees
662   %
663   \prop_gclear_new:c {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
664   \prop_gclear_new:c {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop}
665   \graph_map_vertices_inline:Nn #1 {
666     \prop_put:cnf {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##1}
667     { \graph_get_indegree:Nn #1 {##1} }
668     \int_compare:nT {\graph_get_indegree:Nn #1 {##1} = 0} {
669       \prop_put:cnm {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
670     } }
671
672   %% Main loop
673   %
674   \bool_until_do:nn {\prop_if_empty_p:c {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
675     %% Choose any vertex (\l__graph_topo_key_tl, \l__graph_topo_value_tl)
676     %
677     \__graph_prop_any_key_pop:cN
678     {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
679     \l__graph_topo_key_tl
680     \graph_get_vertex:NVNT #1 \l__graph_topo_key_tl \l__graph_topo_val_tl {
681
682       %% Deduct one from the counter of all affected nodes
683       %% and add all now-empty vertices to source_vertices
684       %
685       \graph_map_outgoing_edges_inline:NVn #1 \l__graph_topo_key_tl {
686         \prop_put:cnf {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2}
687         {\int_eval:n {\prop_item:cn {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2}} }
688         \int_compare:nT {\prop_item:cn {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2} = 0} {
689           \prop_put:cnm {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2}
690         } }
691
692       %% Run the mapping function on the key and value from that vertex
693       %% and manage the nesting depth counter
694       %
695       \int_gincr:N \g__graph_nesting_depth_int
696       \withargs:VVn \l__graph_topo_key_tl \l__graph_topo_val_tl
697       { #2 {##1} {##2} }
698       \int_gdecr:N \g__graph_nesting_depth_int
699     }
700   } }
701 \cs_new_protected:Nn \__graph_prop_any_key_pop:NN {
702   \prop_map_inline:Nn #1 {
703     \tl_set:Nn #2 {##1}
704     \prop_remove:Nn #1 {##1}
705     \prop_map_break:n {\use_none:nmn}
706   }
707   \tl_set:Nn #2 {\q_no_value} % TODO: test
708 }
709 \cs_generate_variant:Nn \__graph_prop_any_key_pop:NN {cN}
710 \cs_generate_variant:Nn \withargs:nnn {VVn}
711 \cs_generate_variant:Nn \graph_map_outgoing_edges_inline:Nnn {NVn}

```

```

712 \cs_generate_variant:Nn \prop_put:Nnn {cnf}
713 \cs_generate_variant:Nn \graph_get_vertex:NnNT {NVNT}
714 \tl_new:N \l__graph_topo_key_tl
715 \tl_new:N \l__graph_topo_val_tl
716 \int_new:N \g__graph_nesting_depth_int

```

Applies the function #2 to all vertex name/value pairs in topological order. The function is supplied with two arguments as trailing brace groups. Assumes that the graph is acyclic (for now).

```

717 \cs_new:Nn \graph_map_topological_order_function:NN {
718   \graph_map_topological_order_tokens:Nn #1 {#2}
719 }

```

Applies the inline function #2 to all vertex name/value pairs in topological order. The inline function is supplied with two arguments: ‘#1’ for the name and ‘#2’ for the value. Assumes that the graph is acyclic (for now).

```

720 \cs_new_protected:Nn \graph_map_topological_order_inline:Nn {
721   \withargs (c) [\uniquecsname] [#2] {
722     \cs_set:Npn ##1 #####1####2 {##2}
723     \graph_map_topological_order_function:NN #1 ##1
724   } }

```

3.12 Transforming Graphs

Set graph #1 to the transitive closure of graph #2.

```

725 \cs_new_protected:Nn \graph_set_transitive_closure:NN {
726   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} { }
727 }
728 \cs_new_protected:Nn \graph_gset_transitive_closure:NN {
729   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} {g}
730 }
731 \cs_new_protected:Nn \graph_set_transitive_closure:NNNn {
732   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} { }
733 }
734 \cs_new_protected:Nn \graph_gset_transitive_closure:NNNn {
735   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} {g}
736 }
737 \cs_new_protected:Nn \__graph_set_transitive_closure:NNNnn {
738   % #1: target
739   % #2: source
740   % #3: combination function with argspec :nnn
741   % #4: default 'old' value
742   \use:c{graph_#5set_eq:NN} #1 #2
743
744   \cs_set:Nn \__graph_edge_combinator:nnn {
745     \exp_not:n { #3 {##1} {##2} {##3} } }
746   \cs_generate_variant:Nn \__graph_edge_combinator:nnn {VVV}
747
748   \graph_map_vertices_inline:Nn #2 {
749     \graph_map_vertices_inline:Nn #2 {

```

```

750 \graph_get_edge:NnnNT #2 {##1} {#####1}
751 \l__graph_edge_value_i_tl {
752 \graph_map_vertices_inline:Nn #2 {
753 \graph_get_edge:NnnNT #2 {##1} {#####1}
754 \l__graph_edge_value_ii_tl {
755 \graph_get_edge:NnnNF #1 {##1} {#####1}
756 \l__graph_edge_value_old_tl {
757 \tl_set:Nn \l__graph_edge_value_old_tl {#4}
758 }
759 \exp_args:NNx \tl_set:No \l__graph_edge_value_new_tl {
760 \__graph_edge_combinator:VVV
761 \l__graph_edge_value_i_tl
762 \l__graph_edge_value_ii_tl
763 \l__graph_edge_value_old_tl
764 }
765 \use:c{graph_#5put_edge:NnnV} #1 {##1} {#####1}
766 \l__graph_edge_value_new_tl
767 } } } } } }
768 \cs_generate_variant:Nn \graph_put_edge:Nnnn {NnnV}
769 \cs_generate_variant:Nn \graph_gput_edge:Nnnn {NnnV}
770 \cs_generate_variant:Nn \tl_to_str:n {o}
771 \tl_new:N \l__graph_edge_value_i_tl
772 \tl_new:N \l__graph_edge_value_ii_tl
773 \tl_new:N \l__graph_edge_value_old_tl
774 \tl_new:N \l__graph_edge_value_new_tl

```

Assume that graph #2 contains no cycles, and set graph #1 to its transitive reduction.

```

775 \cs_new_protected:Nn \graph_set_transitive_reduction:NN {
776 \__graph_set_transitive_reduction:NNn #1 #2 { } }
777 \cs_new_protected:Nn \graph_gset_transitive_reduction:NN {
778 \__graph_set_transitive_reduction:NNn #1 #2 {g} }
779 \cs_new_protected:Nn \__graph_set_transitive_reduction:NNn {
780 % #1: target
781 % #2: source
782 \use:c{graph_#3set_eq:NN} #1 #2
783 \graph_map_vertices_inline:Nn #2 {
784 \graph_map_vertices_inline:Nn #2 {
785 \graph_get_edge:NnnNT #2 {##1} {#####1} \l_tmpa_tl {
786 \graph_map_vertices_inline:Nn #2 {
787 \graph_get_edge:NnnNT #2 {##1} {#####1} \l_tmpa_tl {
788 \use:c{graph_#3remove_edge:Nnn} #1 {##1} {#####1}
789 } } } } }
790 }

```

3.13 Displaying Graphs

We define some additional functions that can display the graph in table-form. This is the option-less version, which delegates to the full version:

```

791 \cs_new_protected:Nn \graph_display_table:N {
792 \graph_display_table:Nn #1 { } }

```

The full version has a second argument accepting options that determine table formatting. We first define those options. Please note that with the standard options, the `xcolor` package is required with the `table` option, because of our use of the `\cellcolor` command.

```

793 \keys_define:nn {lt3graph-display} {
794   row_keys .bool_set:N = \l__graph_display_row_keys_bool,
795   row_keys .initial:n = {true},
796   row_keys .default:n = {true},
797
798   vertex_vals .bool_set:N = \l__graph_display_vertex_vals_bool,
799   vertex_vals .initial:n = {true},
800   vertex_vals .default:n = {true},
801
802   row_keys_format .tl_set:N = \l__graph_format_row_keys_tl,
803   row_keys_format .initial:n = \textbf,
804   row_keys_format .value_required:n = true,
805
806   col_keys_format .tl_set:N = \l__graph_format_col_keys_tl,
807   col_keys_format .initial:n = \textbf,
808   col_keys_format .value_required:n = true,
809
810   vertex_vals_format .tl_set:N = \l__graph_format_vertex_vals_tl,
811   vertex_vals_format .initial:n = \use:n,
812   vertex_vals_format .value_required:n = true,
813
814   edge_vals_format .tl_set:N = \l__graph_format_edge_vals_tl,
815   edge_vals_format .initial:n = \use:n,
816   edge_vals_format .value_required:n = true,
817
818   edge_diagonal_format .tl_set:N = \l__graph_format_edge_diagonal_tl,
819   edge_diagonal_format .initial:n = \cellcolor{black!30!white},
820   edge_diagonal_format .value_required:n = true,
821
822   edge_direct_format .tl_set:N = \l__graph_format_edge_direct_tl,
823   edge_direct_format .initial:n = \cellcolor{green},
824   edge_direct_format .value_required:n = true,
825
826   edge_transitive_format .tl_set:N = \l__graph_format_edge_transitive_tl,
827   edge_transitive_format .initial:n = \cellcolor{green!40!yellow}\tiny(tr),
828   edge_transitive_format .value_required:n = true,
829
830   edge_none_format .tl_set:N = \l__graph_format_edge_none_tl,
831   edge_none_format .initial:n = {},
832   edge_none_format .value_required:n = true
833 }

```

Now we define the function itself. It displays a table showing the structure and content of graph #1. If argument #2 is passed, its options are applied to format the output.

```

834 \cs_new_protected:Nn \graph_display_table:Nn {
835   \group_begin:

```

We process those options passed with #2:


```
836 \keys_set:nn {lt3graph-display} {#2}
```

We populate the top row of the table:

```
837 \tl_put_right:Nn \l__graph_table_content_tl {\hline}
838 \seq_clear:N \l__graph_row_seq
839 \bool_if:NT \l__graph_display_row_keys_bool
840   { \seq_put_right:Nn \l__graph_row_seq {
841     \tl_put_right:Nn \l__graph_table_colspec_tl {|r|} }
842 \bool_if:NT \l__graph_display_vertex_vals_bool
843   { \seq_put_right:Nn \l__graph_row_seq {
844     \tl_put_right:Nn \l__graph_table_colspec_tl {|c|} }
845 \graph_map_vertices_inline:Nn #1 {
846   \tl_put_right:Nn \l__graph_table_colspec_tl {|c}
847   \seq_put_right:Nn \l__graph_row_seq
848     { { \l__graph_format_col_keys_tl {##1} } }
849 }
850 \tl_put_right:Nn \l__graph_table_colspec_tl {|}
851 \tl_put_right:Nx \l__graph_table_content_tl
852   { \seq_use:Nn \l__graph_row_seq {&} }
853 \tl_put_right:Nn \l__graph_table_content_tl
854   { \\hline\hline }
```

We populate the remaining rows:

```
855 \graph_map_vertices_inline:Nn #1 {
856   \seq_clear:N \l__graph_row_seq
857   \bool_if:NT \l__graph_display_row_keys_bool {
858     \seq_put_right:Nn \l__graph_row_seq
859       { { \l__graph_format_row_keys_tl {##1} } } }
860   \bool_if:NT \l__graph_display_vertex_vals_bool {
861     \seq_put_right:Nn \l__graph_row_seq
862       { { \l__graph_format_vertex_vals_tl {##2} } } }
863   \graph_map_vertices_inline:Nn #1 {
```

We start building the vertex cell value. First we distinguish between a direct connection, a transitive connection, and no connection, and format accordingly:

```
864   \graph_get_edge:NnnNTF #1 {##1} {####1} \l_tmpa_tl {
865     \quark_if_no_value:VF \l_tmpa_tl {
866       \tl_set_eq:NN \l__graph_cell_content_tl \l_tmpa_tl
867       \tl_set:Nf \l__graph_cell_content_tl
868         { \exp_args:NV \l__graph_format_edge_direct_tl
869           \l__graph_cell_content_tl } }
870     }{\graph_acyclic_if_path_exist:NnnTF #1 {##1} {####1} {
871       \tl_set_eq:NN \l__graph_cell_content_tl
872         \l__graph_format_edge_transitive_tl
873     }{
874       \tl_set_eq:NN \l__graph_cell_content_tl
875         \l__graph_format_edge_none_tl
876     }}}
```

Secondary formatting comes from cells on the diagonal, i.e., a key compared to itself:

changes	1	0.1.2	General: allowing <code>\graph_map_topo-logical_order...</code> to be nested	1
0.0.9				
General: creation of the documentation	1	0.1.3	General: fixed a bug in <code>\graph_remove_vertex</code> and added a <code>\graph_vertex_count:N</code> function	1
0.1.0				
General: fixed a bug in <code>\graph_(g)put_vertex</code> and <code>\graph_(g)put_edge</code> , which caused their global versions not to work properly	1	0.1.4	General: no longer loading individual l3kernel packages, which leads to an error for recent versions of expl3	1
0.1.1				
General: fixed a similar bug in <code>\graph_(g)put_edges_from</code>	1	0.1.8	General: tracking changes in expl3	1

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		
<code>\</code>	25, 26	11, 11, 11, 15, 16, 17, 19, 20, 21, 21, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 24
B		
<code>\begin</code>	26	<code>\cs_set:Nn</code> 22
bool commands:		<code>\cs_set:Npn</code> 15, 16, 17, 19, 20, 22
<code>\bool_if:NT</code>	25, 25, 25, 25	<code>\cs_set_eq:NN</code> 5, 7, 7, 7, 7
<code>\bool_if:nT</code>	13, 13, 14	
<code>\bool_new:N</code>	26	E
<code>\bool_until_do:nn</code>	21	<code>\end</code> 26
C		exp commands:
<code>\cellcolor</code>	24, 24, 24	<code>\exp_args:Nnv</code> 15
cs commands:		<code>\exp_args:NNx</code> 23
<code>\cs_generate_variant:Nn</code>	7, 9, 9, 9, 10, 11, 11, 11, 11, 13, 15, 16, 16, 17, 17, 18, 18, 19, 20, 21, 21, 21, 22, 22, 22, 23, 23, 23, 26, 26	<code>\exp_args:NV</code> 25, 26, 26
<code>\cs_if_exist:Nf</code>	7	<code>\exp_not:n</code> 22
<code>\cs_if_exist:Np</code>	7	
<code>\cs_if_exist:NT</code>	7	G
<code>\cs_if_exist:NTF</code>	7	graph commands:
<code>\cs_new:Nn</code>	5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 13, 13, 14, 14, 14, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 20, 20, 20, 20, 22	<code>\graph_acyclic_if_path_exist:Nnn</code> 14
<code>\cs_new:Npn</code>	12, 12, 12, 12	<code>__graph_acyclic_if_path_exist:Nnnnn</code> 14, 14
<code>\cs_new_protected:Nn</code>	6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 10, 10, 10, 10,	<code>\graph_acyclic_if_path_exist:NnnTF</code> 25
		<code>\graph_acyclic_if_path_exist_p:Nnn</code> 14
		<code>\l__graph_cell_content_tl</code> 25, 25, 25, 25, 26, 26, 26, 26, 26, 26
		<code>\graph_clear:N</code> 7
		<code>__graph_clear:Nn</code> 7, 7, 7
		<code>\graph_clear_new:N</code> 7
		<code>__graph_clear_new:Nn</code> 7, 7, 7
		<code>\l__graph_display_row_keys_bool</code> 24, 25, 25
		<code>\graph_display_table:N</code> 23

<code>\graph_display_table:Nn</code>	23, 24	<code>\graph_gset_eq:NN</code>	7
<code>\l__graph_display_vertex_vals_-</code> <code>bool</code>	24, 25, 25	<code>\graph_gset_transitive_closure:NN</code>	22
<code>__graph_edge_combinator:nnn</code>	22, 22	<code>\graph_gset_transitive_closure:NNNn</code>	22
<code>__graph_edge_combinator:VVV</code>	23	<code>\graph_gset_transitive_reduction:NN</code>	23
<code>\l__graph_edge_data_tl</code>	9, 9, 9, 9, 10, 11	<code>\graph_if_cyclic:N</code>	13
<code>\l__graph_edge_value_i_tl</code>	23, 23, 23	<code>__graph_if_cyclic:Nnn</code>	13, 13
<code>\l__graph_edge_value_ii_tl</code>	23, 23, 23	<code>\graph_if_edge_exist:Nnn</code>	12
<code>\l__graph_edge_value_new_tl</code>	23, 23, 23	<code>\graph_if_exist:NF</code>	7
<code>\l__graph_edge_value_old_tl</code>	23, 23, 23, 23	<code>\graph_if_exist:Np</code>	7
<code>__graph_empty_set</code>	5, 12, 12, 12, 12	<code>\graph_if_exist:NT</code>	7
<code>__graph_for_each_prop_datatype:n</code>	6, 6, 7, 7	<code>\graph_if_exist:NTF</code>	6, 7
<code>\l__graph_format_col_keys_tl</code>	24, 25	<code>\graph_if_exists:NF</code>	7
<code>\l__graph_format_edge_diagonal_-</code> <code>tl</code>	24, 26	<code>\graph_if_vertex_can_reach_-</code> <code>cycle:NnF</code>	12
<code>\l__graph_format_edge_direct_tl</code>	24, 25	<code>__graph_if_vertex_can_reach_-</code> <code>cycle:Nnn</code>	12
<code>\l__graph_format_edge_none_tl</code>	24, 25	<code>__graph_if_vertex_can_reach_-</code> <code>cycle:NnnF</code>	12
<code>\l__graph_format_edge_transitive_-</code> <code>tl</code>	24, 25	<code>__graph_if_vertex_can_reach_-</code> <code>cycle:Nnnnn</code>	12, 13
<code>\l__graph_format_edge_vals_tl</code>	24, 26	<code>__graph_if_vertex_can_reach_-</code> <code>cycle:NnnT</code>	12
<code>\l__graph_format_row_keys_tl</code>	24, 25	<code>__graph_if_vertex_can_reach_-</code> <code>cycle:NnnTF</code>	12
<code>\l__graph_format_vertex_vals_tl</code>	24, 25	<code>\graph_if_vertex_can_reach_-</code> <code>cycle:NnT</code>	12
<code>\l__graph_from_value_tl</code>	8, 9, 9	<code>\graph_if_vertex_can_reach_-</code> <code>cycle:NnTF</code>	12
<code>\graph_gclear:N</code>	7	<code>\graph_if_vertex_can_reach_-</code> <code>cycle:NnTF</code>	12
<code>\graph_gclear_new:N</code>	7	<code>\graph_if_vertex_can_reach_-</code> <code>cycle_p:Nn</code>	12, 13
<code>__graph_get_cycle:NN</code>	13	<code>__graph_if_vertex_can_reach_-</code> <code>cycle_p:Nnn</code>	12, 13
<code>\graph_get_cycle:NN</code>	13	<code>__graph_if_vertex_can_reach_-</code> <code>cycle_p:Nno</code>	13
<code>__graph_get_cycle:NNTF</code>	13	<code>\graph_if_vertex_exist:Nn</code>	12
<code>\graph_get_degree:Nn</code>	15	<code>\graph_if_vertex_exist:NnT</code>	11, 11
<code>\graph_get_edge:NnnN</code>	11	<code>__graph_key:n</code>	6
<code>\graph_get_edge:NnnNF</code>	8, 23	<code>__graph_key:nn</code>	6, 9, 9, 9, 9, 10, 11, 11, 11, 12, 12
<code>\graph_get_edge:NnnNT</code>	10, 23, 23, 23, 23	<code>__graph_key:nnn</code>	6
<code>\graph_get_edge:NnnNTF</code>	25	<code>__graph_key:nnnn</code>	6
<code>\graph_get_indegree:Nn</code>	15, 15, 21, 21	<code>__graph_key:nnnnn</code>	6
<code>\graph_get_outdegree:Nn</code>	14, 15	<code>\graph_map_edges_function:NN</code>	16, 16
<code>\graph_get_vertex:NnN</code>	11	<code>__graph_map_edges_function_-</code> <code>aux:Nnn</code>	16, 16
<code>\graph_get_vertex:NnNT</code>	8, 10, 22	<code>__graph_map_edges_function_-</code> <code>aux:Nnnn</code>	16, 16
<code>\graph_get_vertex:NnNTF</code>	8, 8	<code>__graph_map_edges_function_-</code> <code>aux:Nnnv</code>	16
<code>\graph_get_vertex:NVNT</code>	21	<code>\graph_map_edges_inline:Nn</code>	11, 16
<code>\graph_gput_edge:Nnn</code>	8	<code>\graph_map_edges_tokens:Nn</code>	16
<code>\graph_gput_edge:Nnnn</code>	8, 23		
<code>\graph_gput_edges_from:NN</code>	11		
<code>__graph_gput_edges_from:NNn</code>	11, 11, 11		
<code>\graph_gput_vertex:Nn</code>	8		
<code>\graph_gput_vertex:Nnn</code>	8		
<code>\graph_gremove_edge:Nnn</code>	10		
<code>\graph_gremove_vertex:Nn</code>	9		

_graph_map_edges_tokens_-		\graph_map_successors_inline:Nnn	
aux:nnn	16, 16	14, 20
_graph_map_edges_tokens_-		\graph_map_successors_tokens:Nnn	19
aux:nnnn	16, 16	_graph_map_successors_tokens_-	
_graph_map_edges_tokens_-		aux:nnn	19, 19
aux:nnnv	16	_graph_map_successors_tokens_-	
\graph_map_incoming_edges_-		aux:Nnnnn	19, 19
function:NnN	17, 17	_graph_map_successors_tokens_-	
_graph_map_incoming_edges_-		aux:Nnnnnn	19, 19
function_aux:nNnn	17, 17	_graph_map_successors_tokens_-	
_graph_map_incoming_edges_-		aux:nnv	19
function_aux:nNnnn	17, 17	\graph_map_topological_order_-	
_graph_map_incoming_edges_-		function:NN	22, 22
function_aux:nNnnv	17	\graph_map_topological_order_-	
\graph_map_incoming_edges_-		inline:Nn	22
inline:Nnn	10, 17	\graph_map_topological_order_-	
\graph_map_incoming_edges_-		tokens:Nn	21, 22
tokens:Nnn	16	\graph_map_vertices_function:NN	
_graph_map_incoming_edges_-		15, 15
tokens_aux:nnnn	16, 16	\graph_map_vertices_inline:Nn	15,
_graph_map_incoming_edges_-		21, 22, 22, 23, 23, 23, 23, 25, 25, 25	
tokens_aux:nnnnn	17, 17	\graph_map_vertices_tokens:Nn	13, 15
_graph_map_incoming_edges_-		_graph_map_vertices_tokens_-	
tokens_aux:nnnnv	16	aux:nnn	15, 15
\graph_map_outgoing_edges_-		_graph_map_vertices_tokens_-	
function:NnN	18, 19	aux:nnv	15
_graph_map_outgoing_edges_-		\g_graph_nesting_depth_int	
function_aux:nNnn	18, 18	21, 21,
_graph_map_outgoing_edges_-		21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 22	
function_aux:nNnnn	18, 18	\graph_new:N	6, 7
_graph_map_outgoing_edges_-		_graph_prop_any_key_pop:cN	21
function_aux:nNnnv	18	_graph_prop_any_key_pop:NN	21, 21
\graph_map_outgoing_edges_-		\g_graph_prop_data_types_seq	
inline:Nnn	10, 19, 21	6, 6, 6
\graph_map_outgoing_edges_-		_graph_ptr_gnew:Nn	6
inline:NVn	21	_graph_ptr_new:Nn	6
\graph_map_outgoing_edges_-		\graph_put_edge:Nnn	8
tokens:Nnn	12, 14, 18	\graph_put_edge:Nnnn	8, 23
_graph_map_outgoing_edges_-		_graph_put_edge:Nnnnn	8, 8, 8, 8, 8
tokens_aux:nnnn	18, 18	\graph_put_edges_from:NN	11
_graph_map_outgoing_edges_-		\graph_put_vertex:Nn	8
tokens_aux:nnnnn	18, 18	\graph_put_vertex:Nnn	8
_graph_map_outgoing_edges_-		_graph_put_vertex:Nnnn	8, 8, 8, 8, 8
tokens_aux:nnnnv	18	\graph_remove_edge:Nnn	10
\graph_map_successors_function:NnN		_graph_remove_edge:Nnnn	10, 10, 10
.....	20, 20	\graph_remove_vertex:Nn	9
_graph_map_successors_function_-		_graph_remove_vertex:Nnn	9, 9, 10
aux:Nnn	20, 20	\l_graph_row_seq	
_graph_map_successors_function_-		25, 25, 25, 25, 25, 25, 25, 26, 26, 26	
aux:NnNnn	20, 20	_graph_set_cons:nn	5, 13
_graph_map_successors_function_-		\graph_set_eq:NN	7
aux:NnNnnn	20, 20	_graph_set_eq:NNn	7, 7, 7
_graph_map_successors_function_-		_graph_set_if_in:nn	5
aux:Nnv	20	_graph_set_if_in:p:nn	13

R	
recursion commands:	
\q_recursion_tail	5
\RequirePackage	5, 5, 5, 5
S	
seq commands:	
\seq_clear:N	13, 25, 25
\seq_if_in:NnTF	14
\seq_map_inline:Nn	6
\seq_new:N	6, 26
\seq_put_right:Nn ..	25, 25, 25, 25, 25
\seq_put_right:NV	26
\seq_set_from_clist:Nn	6
\seq_use:Nn	25, 26
str commands:	
\str_if_eq:nnT	17, 17, 18, 18, 19, 20, 26
\str_if_eq_p:nn	14
\str_tail:n	6
T	
\textbf	24, 24
\tiny	24
tl commands:	
\tl_gset:cn	6
\tl_gset:Nn	6
\tl_new:c	6, 6
\tl_new:N	6, 8, 9, 9, 9, 10, 11, 22, 22, 23, 23, 23, 23, 26, 26, 26
\tl_put_right:Nn	25, 25, 25, 25, 25, 25, 26
\tl_put_right:Nx	25, 26
\tl_set:cn	6
\tl_set:Nf	6, 25, 26, 26
\tl_set:Nn	6, 21, 21, 23
\tl_set:No	23
\tl_set:Nv	11, 12
\tl_set_eq:NN	25, 25, 25
\tl_to_str:n	9, 9, 9, 9, 23
\tl_trim_spaces:f	6
\tl_trim_spaces:n	7
tmpa commands:	
\l_tmpa_tl	8, 8, 23, 23, 25, 25, 25
U	
\uniquecsname ..	6, 6, 15, 16, 17, 19, 20, 22
use commands:	
\use:c ..	7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 22, 23, 23, 23
\use:n	24, 24
\use_i:mn	13, 13, 14
\use_none:nnn	21, 22, 22
W	
\withargs	6, 6, 15, 16, 17, 19, 20, 22
withargs commands:	
\withargs:nnn	9, 21, 26
\withargs:VVn	9, 21, 26