

# Compiler

Klaus Grue

GRD-2009-11-13.UTC:10:36:36.029114

## Contents

<b>1</b>	<b>The lgc compiler</b>	<b>4</b>
1.1	Frontend language	4
1.2	File extensions	5
1.3	Files	5
1.4	General functions	6
1.5	Event handling functions	6
1.6	State machine	6
1.7	Machine	7
1.8	State entries	7
1.9	Reading of configuration files	9
1.10	Parameter processing	11
1.11	Query processing	15
<b>2</b>	<b>Time functions</b>	<b>18</b>
2.1	Time schemes	18
2.2	TAI	18
2.3	UTC	19
2.4	MJD	20
2.5	GRD	21
2.6	LGT	22
2.7	GUTC	23
2.8	Unix time	24
2.9	The purpose of time stamps	25
2.10	Parameter names	25
2.11	Constants	26
2.12	Conversion from GRD to MJD	27
2.13	Conversion from MJD to GRD	27
2.14	GRD Parsing functions	28
2.15	Conversion from reference to Logiweb time	30
2.16	Conversion from Logiweb time to printed representation	30
2.17	Conversion from Logiweb time to MJD/TAI	30
2.18	Conversion from Logiweb time to GRD/UTC	31
2.19	Conversion from Unix time to Logiweb time	32

<b>3</b>	<b>Message generation</b>	<b>33</b>
3.1	State entries . . . . .	33
3.2	Integer input/output . . . . .	33
3.3	Verbosity levels . . . . .	35
3.4	Message stacking . . . . .	35
3.5	Message reporting . . . . .	36
3.6	Unconditional errors . . . . .	38
3.7	Progress messages . . . . .	38
<b>4</b>	<b>Lexical analysis</b>	<b>39</b>
4.1	State entries . . . . .	40
4.2	Lexical analysis main functions . . . . .	41
4.3	Escape sequences . . . . .	42
4.4	Space trimming . . . . .	43
4.5	Space contraction . . . . .	43
4.6	Space trimming of text containing character positions . . . . .	44
4.7	Space contraction of text containing character positions . . . . .	44
4.8	Lexical analysis . . . . .	45
4.9	Add positions to text . . . . .	46
4.10	Replace newline sequences with newline characters . . . . .	46
4.11	Remove comments . . . . .	47
4.12	Parse escape sequences . . . . .	48
4.13	Collect structures . . . . .	48
4.14	Space parsing . . . . .	49
4.15	Page parsing . . . . .	49
4.16	Reference parsing . . . . .	49
4.17	Definition parsing . . . . .	50
4.18	String parsing . . . . .	51
4.19	Extract result of lexical analysis . . . . .	52
4.20	Include processing . . . . .	53
<b>5</b>	<b>Loading</b>	<b>53</b>
5.1	Load processing . . . . .	53
5.2	State entries . . . . .	54
5.3	Loading in more detail . . . . .	55
5.4	Top level function . . . . .	57
5.5	Constants . . . . .	58
5.6	Auxiliary functions . . . . .	59
5.7	Message generators . . . . .	60
5.8	Requesting pages . . . . .	61
5.9	Receiving pages . . . . .	63
5.10	Codifying loaded pages . . . . .	66
5.11	Trisecting . . . . .	67
5.12	Codifying . . . . .	69
5.13	Unpacking . . . . .	70
5.14	Initializing . . . . .	71

5.15	Macro expanding	72
5.16	Harvesting	72
<b>6</b>	<b>Grammars</b>	<b>73</b>
6.1	Tokens	73
6.2	Source grammars	75
6.3	Associativities	75
6.4	Constructs	76
6.5	Qualified constructs	77
6.6	Specifying qualification in source files	79
6.7	Grammar ambiguity	80
6.8	Trie grammars	80
6.9	Grammar nodes	80
6.10	Grammar construction	81
6.11	Constructs from referenced pages	82
6.12	Default names	84
6.13	Constructs from source text	85
<b>7</b>	<b>Parser</b>	<b>87</b>
7.1	Token lists	87
7.2	Linear parse trees	88
7.3	Left parse trees	88
7.4	Representation of left parse trees	89
7.5	Return value	89
7.6	Restrictions on left parse trees	89
7.7	Functions for invoking charge rules	90
7.8	Main charge functions	91
7.9	Charge handling of brackets and braces	92
7.10	Charge handling of strings	95
7.11	Autogeneration of name and charge definitions	97
7.12	Vectorizing	98
7.13	Invocation of the parser	100
7.14	Definition of the parser	102
7.15	Message generators	104
7.16	Codification of parsed page	107
<b>8</b>	<b>Rendering</b>	<b>107</b>
8.1	Layout of rendering	107
8.2	State entries	108
8.3	Messages	108
8.4	Translation of terms to Logiweb source	109
8.5	Translation of terms to Logiweb source	110
8.6	General html constructors	111
8.7	Rendering	114
8.8	Rendering directory	114
8.9	Rendering of links	115

8.10	Rendering of non-html	116
8.11	Rendering of index	117
8.12	Rendering of extract	117
8.13	Rendering of bibliography	118
8.14	Rendering of definitions	119
8.15	Rendering of charges	121
8.16	Verification	123
8.17	HTML rendering of diagnose	124
8.18	Dumping to cache	124
<b>9</b>	<b>User rendering</b>	<b>125</b>
9.1	Overview	125
9.2	Variable names	126
9.3	Main user rendering functions	126
9.4	Functions for executing individual rendering events	128
9.5	Safety check filename	130
9.6	Invocation of user rendering	131
9.7	Default rendering of executables	134
9.8	Rendering of lgwinclude file	135
9.9	Rendering compiler	136
9.10	Rendering evaluator	137
9.11	Use rendering	138
9.12	Show rendering	139
9.13	Rendering based on name	140
9.14	Show rendering of strings	141
<b>10</b>	<b>Test cases</b>	<b>144</b>
10.1	Lexical Analysis	144
10.2	Splicing	147
10.3	Auxiliary definitions	148
10.4	Trisecting of lgw files	148
10.5	Test of conversion from GRD to MJD	148
10.6	Test of conversion from MJD to GRD	150
10.7	Test of GRD parsing functions	152
10.8	Test of conversion from Logiweb time to MJD/TAI	153
10.9	Test of conversion from Logiweb time to GRD/UTC	154
10.10	Test of conversion from Unix time to Logiweb time	161

# 1 The lgc compiler

## 1.1 Frontend language

The Logiweb compiler (lgc) comprises a *frontend*, a *codifier*, and a *renderer*.

The frontend takes a Logiweb *source* file as input. Source files typically have extension .lgs; they are human readable and human editable files expressed in the Logiweb *frontend language*.

The frontend produces a Logiweb *vector* file as output. Vector files typically have extension `.lgw`; they are compact, binary files suited for transmission over the Internet. We shall refer to the contents of a vector file as a Logiweb *vector*, regardless of whether the vector is retrieved from a file, retrieved over a network, or is passed in memory from the frontend.

The codifier takes a vector as input and produces a Logiweb *rack* file as output. Rack files typically have extension `.lgr`; they are big, binary files suited for storing in the local file system. A file or memory structure which allows to look up racks given their references is termed a Logiweb *cache*. We shall refer to the contents of a rack file as a Logiweb *rack*, regardless of how it is retrieved.

The renderer takes a rack as input and produces a Logiweb *rendering* as output. A rendering is a directory structure of interlinked html, pdf, and other files suited for viewing in a browser.

Any Logiweb compiler must contain a frontend, a codifier, and a renderer. Furthermore, the codifier of any Logiweb compiler must do exactly as the codifier defined in the following. The frontend and renderer, however, are replacable.

As an example, one may design a language different from the Logiweb frontend language and write a frontend for that language. One could even imagine a wysiwyg Logiweb frontend. The only requirement to a frontend is that it must produce a Logiweb vector as output.

Any backend should take a Logiweb rack as input and produce some sort of rendering as output.

## 1.2 File extensions

The Logiweb compiler uses the following file extensions:

**.lgw** Logiweb vector (lgw for LoGiWeb).

**.lgs** Logiweb source text (s for source).

**.lgr** Logiweb rack (r for rack).

**.lgu** Indirection (u for URL).

**.lgp** Reserved for Logiweb reference (p for pointer).

**.lgo** Reserved for link to Logiweb rendering (o for output).

## 1.3 Files

`/etc/logiweb/lgc.conf` Default location for site configuration file.

`$HOME/.logiweb/lgc.conf` Default location for a user configuration file.

`$HOME/.logiweb/logiweb/` Default location for output and racks.

`/usr/bin/lgwam` Default location of Logiweb abstract machine.

`/usr/bin/lgc` Default location of Logiweb compiler.

`/usr/man` Default location of Logiweb man pages.

## 1.4 General functions

$[[u \stackrel{\bullet\bullet}{=} v] \stackrel{\bullet}{=} [u \stackrel{\bullet}{=} v]]$

## 1.5 Event handling functions

$[\text{lgc-get-events} ( s ) \stackrel{\bullet\bullet}{=} s[\text{'events'}]]$

Return the list of events to be executed. The list is in reverse order.

$[\text{lgc-set-events} ( s , E ) \stackrel{\bullet\bullet}{=} s[\text{'events'} \rightarrow E]]$

Set the list of events to the list  $E$ .

$[\text{lgc-clr-events} ( s ) \stackrel{\bullet\bullet}{=} \text{lgc-set-events} ( s , \top )]$

Clear the list of events.

$[\text{lgc-push-event} ( s , e ) \stackrel{\bullet\bullet}{=} \text{push} ( s , \text{'events'} , e )]$

Add an event  $e$  to list  $E$  of events. When executed,  $e$  will be executed after the events in  $E$ .

$[\text{lgc-do-events} ( s ) \stackrel{\bullet\bullet}{=} \text{reverse} ( \text{lgc-get-events} ( s ) )]$

Push the event  $e$  onto the list of events, then reverse the list.

$[\text{lgc-exec} ( f ) \stackrel{\bullet}{=} \text{exec request} ( \top , \text{map} ( \lambda s . \lambda x . f ) " \text{lgc-clr-events} ( s )^M )]$

Forge an execute event for invoking  $f$ .

$[\text{lgc-exec-events} ( s , f ) \stackrel{\bullet}{=} \text{reverse} ( \text{lgc-exec} ( f ) :: \text{lgc-get-events} ( s ) )]$

Push an execute event onto the list of events, then reverse the list.

## 1.6 State machine

At top level, `lgc` is a state machine with the following states:

```
lgc-config-1 ( x )
lgc-config-2 ( x , s )
lgc-config-3 ( x , s )
lgc-config-5 ( x , s )
lgc-config-6 ( x , s )
lgc-lex-2 ( x , s )
lgc-lex-3 ( x , s )
lgc-include-2 ( x , s )
```

```

lgc-load-receive ( x , s )
lgc-load-codify1 ( x , s )
lgc-grammar1 ( x , s )
lgc-parse1 ( x , s )
lgc-parse-codify-lgw ( x , s )
lgc-parse6 ( x , s )
lgc-render ( x , s )
lgc-render-verify ( x , s )
lgc-render-dump ( x , s )
lgc-render-user ( x , s )
lgc-render-user2 ( x , s )

```

## 1.7 Machine

The following statements define the main program of the Logiweb compiler.

```

[‘lgc’  $\xrightarrow{\text{execute}}$ 
  (lgc-main, ‘siteconfig=/etc/logiweb/lgc.conf’)]
  Dump the lgc compiler lgc-main under the name of lgc and with the
  given compiled in default.

```

```

[lgc-main  $\equiv$  map (  $\lambda x$ .lgc-config-1 ( x ) )]
  As the first activity, read configuration files.

```

## 1.8 State entries

During parameter processing, the compiler builds up a state  $s$  with the following entries:

- $s$ ['init'] The initial event containing command line arguments, environment variables, and compiled in defaults.
- $s$ ['nsiteconfig'] Tilde expanded name of site configuration file.
- $s$ ['siteconfig'] Contents of site configuration file (if any).
- $s$ ['nuserconfig'] Tilde expanded name of user configuration file.
- $s$ ['userconfig'] Contents of user configuration file (if any).
- $s$ ['parameters'] Array of all parameters.
- $s$ ['verbose'] Verbosity level as a cardinal.

The array  $t$  of parameters has the following entries:

- $t$ ['source'] The name of the source file.

- `t[‘leap’]` List of leap second definitions like ‘GRD-1972-06-30+1’ which indicates that Gregorian Date 1972-06-30 ended with one, positive leap second.
- `t[‘siteconfig’]` The location of the site configuration file. Not tilde-expanded.
- `t[‘userconfig’]` The location of the user configuration file. Not tilde-expanded.
- `t[‘path’]` List of locations to look up pages given their reference. As an example, an item of value ‘~/logiweb/logiweb/~/rack.lgr’ makes the compiler look in the given location under the users home catalog where the rightmost colon is replaced by the reference of the page in mixed endian hexadecimal.
- `t[‘namepath’]` List of locations to look up pages given their name. The rightmost colon is replaced by the name of the referenced page when translating ‘""R’ directives.
- `t[‘rendering’]` Location of rendering output and racks. The rightmost colon is replaced by the reference of the page.
- `t[‘link’]` List of locations for storing links to the rendering output. The rightmost colon is replaced by the name of the source file of the page (with the suffix removed, if any).
- `t[‘newline’]` Host newline (CR, LF, CRLF, or LFCR).
- `t[‘script’]` Script headline.
- `t[‘verbose’]` Verbosity, c.f. Section 3.3.
- `t[‘options’]` When unequal to ‘no’: print the values of all options to standard output and exit.
- `t[‘help’]` When unequal to ‘no’: print help message one to standard output and exit.
- `t[‘help2’]` When unequal to ‘no’: print help message two to standard output and exit.
- `t[‘help3’]` When unequal to ‘no’: print help message three to standard output and exit.
- `t[‘version’]` When unequal to ‘no’: print version to standard output and exit.
- `t[‘license’]` When unequal to ‘no’: print license information to standard output and exit.

## 1.9 Reading of configuration files

Parameters are collected from the following sources:

- Static defaults which are defined in the code.
- Dynamic defaults like `siteconfig=/etc/logiweb/lgc.conf` which are in the code but are easy to modify after compilation.
- Parameters defined in a site configuration file.
- Parameters defined in a user configuration file.
- Environment variables.
- Command line parameters.

The sources are shown in order of precedence in the sense that command line parameters override environment variables and so on.

All parameters can be defined in any of the locations. As an exception, however, the location of the site configuration file cannot be defined in the site or user configuration file and the location of the user configuration file cannot be defined in the user configuration file.

Each parameter is equal to a list of values. Environment variables merely allow to define one-element lists. All other sources allow to set a parameter to a one-element list or to add an element at the beginning or end of the current list.

Configuration files may contain lines like the following:

```
# This is a comment
uuu=xxx
vvv+yyy
www++zzz
```

The lines above set parameter `uuu` to `xxx`, adds `yyy` in front of `vvv`, and adds `zzz` at the end of `www`.

Command lines may contain arguments like the following:

```
--uuu=xxx --vvv+yyy --www++zzz
```

The command line arguments above have the same effect as the configuration file above.

Command line parameters typically have short forms. If `u`, `v`, and `w` are the short forms of `uuu`, `vvv`, and `www`, then one may write:

```
-u xxx +v yyy ++w zzz
```

The two given command lines have the same effect.

Configuration files and command line parameters are processed in the reading direction. As an example, `--vvv+aaa --vvv+bbb` adds first `aaa` and then `bbb` in front of `vvv` so that `bbb` ends up occurring in front of `aaa`.

Dynamic defaults, environment variables, and command line arguments are embedded in the parameter  $x$  of `lgc-config-1 ( x )` below. The functions in the following read in the site and user configuration files.

```
[lgc-config-1 ( x ) ≡  
  let s = T['init'→x] in  
  let t = lgc-process-parameters ( s ) in  
  let n = lgc-tilde-expand ( t['siteconfig']h , t ) in  
  let s = s['nsiteconfig'→n] in ⟨  
    extend request ( lgcio ( T ) , lgcio-interface ) ,  
    unixTime ,  
    fileGetCwd ,  
    fileType ( n ) ,  
    lgc-exec ( lgc-config-2 ( x , s ) ) ⟩ ] ]  
  Ask if the site configuration file exists.
```

```
[lgc-config-2 ( x , s ) ≡  
  let ⟨T, ⟨T, f⟩, ⟨T, d⟩, u⟩ = x in  
  let s = s['time'→parse-unixTime ( u )] in  
  let s = s['cwd'→d] in  
  if fh ≠ 258 then lgc-config-4 ( s ) else ⟨  
    fileRead ( s['nsiteconfig'] ) ,  
    lgc-exec ( lgc-config-3 ( x , s ) ) ⟩ ] ]  
  If the site configuration file exists, read it. Otherwise, pretend that  
  the site configuration file is empty and go on to the user configuration  
  file.
```

```
[lgc-config-3 ( x , s ) ≡  
  let ⟨T, ⟨T, f⟩⟩ = x in  
  let s = s['siteconfig'→f] in lgc-config-4 ( s ) ] ]  
  Store the site configuration file and go on to the user configuration  
  file.
```

```
[lgc-config-4 ( s ) ≡  
  let t = lgc-process-parameters ( s ) in  
  let n = lgc-tilde-expand ( t['userconfig']h , t ) in  
  let s = s['nuserconfig'→n] in ⟨  
    fileType ( n ) ,  
    lgc-exec ( lgc-config-5 ( x , s ) ) ⟩ ] ]  
  Ask if the user configuration file exists.
```

```
[lgc-config-5 ( x , s ) ≐
  let ⟨T, ⟨T, f⟩⟩ = x in
  if fh ≠ 258 then lgc-process-query ( s ) else ⟨
  fileRead ( s[‘nuserconfig’] ),
  lgc-exec ( lgc-config-6 ( x , s ) )⟩]
```

If the user configuration file exists, read it. Otherwise, pretend that the user configuration file is empty and go on to query processing.

```
[lgc-config-6 ( x , s ) ≐
  let ⟨T, ⟨T, f⟩⟩ = x in
  let s = s[‘userconfig’→f] in lgc-process-query ( s )]
```

Store the site configuration file and go on to query processing.

```
[lgc-tilde-expand ( n , t ) ≐
  if n then T else
  if vector-head ( n ) ≠ - NULL then n else
  t[home]::vector-tail ( n )]
```

## 1.10 Parameter processing

This section defines `lgc-process-parameters ( s )` which converts a state `s` into an array which maps parameter names to parameter value lists. The state `s` must contain an initial event and may contain a site and a user configuration file.

```
[lgc-default-leap ≐ ⟨
  ‘GRD-2008-12-31+1’,
  ‘GRD-2005-12-31+1’,
  ‘GRD-1998-12-31+1’,
  ‘GRD-1997-06-30+1’,
  ‘GRD-1995-12-31+1’,
  ‘GRD-1994-06-30+1’,
  ‘GRD-1993-06-30+1’,
  ‘GRD-1992-06-30+1’,
  ‘GRD-1990-12-31+1’,
  ‘GRD-1989-12-31+1’,
  ‘GRD-1987-12-31+1’,
  ‘GRD-1985-06-30+1’,
  ‘GRD-1983-06-30+1’,
  ‘GRD-1982-06-30+1’,
  ‘GRD-1981-06-30+1’,
  ‘GRD-1979-12-31+1’,
  ‘GRD-1978-12-31+1’,
  ‘GRD-1977-12-31+1’,
  ‘GRD-1976-12-31+1’,
  ‘GRD-1975-12-31+1’,
```

```

‘GRD-1974-12-31+1’,
‘GRD-1973-12-31+1’,
‘GRD-1972-12-31+1’,
‘GRD-1972-06-30+1’)]

```

Define the static default for the leap parameter.

```

[lgc-default-options ≡ T[
  ‘leap’→lgc-default-leap][
  ‘siteconfig’→⟨‘/etc/logiweb/lgc.conf’⟩][
  ‘userconfig’→⟨‘~/logiweb/lgc.conf’⟩][
  ‘path’→⟨‘~/logiweb/logiweb/:/rack.lgr’⟩][
  ‘rendering’→⟨‘~/logiweb/logiweb/:/’⟩][
  ‘link’→⟨‘:’, ‘~/logiweb/name/:/’⟩][
  ‘namepath’→⟨‘~/logiweb/name/:/rack.lgr’⟩][
  ‘newline’→⟨‘LF’⟩][
  ‘script’→⟨‘#!/usr/bin/lgwam script’⟩][
  ‘verbose’→⟨‘3’⟩][
  ‘options’→⟨‘no’⟩][
  ‘help’→⟨‘no’⟩][
  ‘help2’→⟨‘no’⟩][
  ‘help3’→⟨‘no’⟩][
  ‘version’→⟨‘no’⟩][
  ‘license’→⟨‘no’⟩]]

```

Define static defaults for all parameters.

```

[lgc-process-argv ( v , t ) ≡
  if v ∈ A then t else
  let a :: v = v in
  if vector-prefix ( a , 2 ) = ‘-’ then
  let l = lgc-process-long-argv ( vector-suffix ( a , 2 ) , t ) in
  lgc-process-argv ( v , l ) else
  if vector-prefix ( a , 1 ) = ‘-’ then
  lgc-process-short-argv ( vector-suffix ( a , 1 ) , ‘-’ , v , t ) else
  if vector-prefix ( a , 2 ) = ‘++’ then
  lgc-process-short-argv ( vector-suffix ( a , 2 ) , ‘++’ , v , t ) else
  if vector-prefix ( a , 1 ) = ‘+’ then
  lgc-process-short-argv ( vector-suffix ( a , 1 ) , ‘+’ , v , t ) else
  lgc-process-argv ( v , t[‘source’→⟨a⟩] )

```

Process the command line arguments given in  $v$  and add them to the array  $t$ . It is assumed that  $v$  is the list of genuine arguments, i.e. that  $\text{argv}[0]$  has been removed.

```

[lgc-argv-short-to-long ≡ T[
  ‘s’→‘siteconfig’][
  ‘u’→‘userconfig’][
  ‘p’→‘path’][

```

```

'n'→'namepath'[]
'r'→'rendering'[]
'l'→'link'[]
'o'→'option'[]
'h'→'help'[]
'H'→'help2'[]
'v'→'verbose'[]

```

Define the mapping from short to long parameter names. Internally in the compiler, parameters are always stored under their long name.

```

[lgc-process-short-argv ( k , a , v , t ) **
  let k' = lgc-argv-short-to-long[k] in
  let k = if k' ≠ T then k' else k in
  lgc-process-argv ( v† , lgc-process-assignment ( k , a , vh , t ) )]

```

Process the short option with keyword  $k$  and assignment operator  $a$ . The assignment operator may be  $-$  or  $+$  or  $++$ .

```

[lgc-split-long-arg ( a ) **
  if a = ' then 0 else
  if vector-prefix ( a , 1 ) = '=' then 0 else
  if vector-prefix ( a , 1 ) = '+' then 0 else
  1 + lgc-split-long-arg ( vector-suffix ( a , 1 ) )]

```

Return the index of the end of the keyword. This is not very efficient, but who cares about efficiency in command line handling?

```

[lgc-argv-downcase ( a ) **
  bt2vector ( lgc-argv-downcase1 ( vector2byte* ( a ) ) )]
  Change A..Z to a..z in the vector  $a$ .

```

```

[lgc-argv-downcase1 ( a ) **
  if a ∈ A then T else let c :: a = a in
  lgc-argv-downcase2 ( c ) :: lgc-argv-downcase1 ( a )]
  Change A..Z to a..z in the list  $a$  of bytes.

```

```

[lgc-argv-downcase2 ( c ) **
  if c < 'A' - NULL then c else
  if c > 'Z' - NULL then c else
  c - 'A' + 'a']
  Change A..Z to a..z.

```

```

[lgc-process-long-argv ( a , t ) **
  let p = lgc-split-long-arg ( a ) in
  let k = lgc-argv-downcase ( vector-prefix ( a , p ) ) in
  let a = vector-suffix ( a , p ) in
  if a = ' then t[k→T] else
  if vector-prefix ( a , 1 ) = '=' then

```

```

lgc-process-assignment ( k , '-' , vector-suffix ( a , 1 ) , t ) else
if vector-prefix ( a , 2 ) = '++' then
lgc-process-assignment ( k , '++' , vector-suffix ( a , 2 ) , t ) else
lgc-process-assignment ( k , '+' , vector-suffix ( a , 1 ) , t )]]

```

Split the long option  $a$  into a keyword, an assignment operator, and a value and execute the associated assignment.

Process the long option  $a$ .

```

[[lgc-process-assignment ( k , a , v , t ) ≡
if a = '-' then t[k→⟨v⟩] else
if a = '+' then t[k→v::t[k]] else
t[k→append ( t[k] , ⟨v⟩ )]]]

```

Assign the value  $v$  to the keyword  $k$  in the array  $t$  using the assignment operator  $a$ .

```

[[lgc-process-env ( v , t ) ≡
if v ∈ A then t else
let a::v = v in
if lgc-argv-downcase ( vector-prefix ( a , 4 ) ) = 'lgc_' then
let l = lgc-process-long-argv ( vector-suffix ( a , 4 ) , t ) in
lgc-process-env ( v , l ) else
if lgc-argv-downcase ( vector-prefix ( a , 5 ) ) = 'home=' then
lgc-process-env ( v , lgc-process-long-argv ( a , t ) ) else
lgc-process-env ( v , t )]]]

```

Process the environment arguments given in  $v$  and add them to the array  $t$ .

```

[[lgc-process-lines ( l , t ) ≡
if l ∈ A then t else let a::l = l in
let t = lgc-process-long-argv ( a , t ) in
lgc-process-lines ( l , t )]]]

```

Process the configuration file lines given in  $v$  and add them to the array  $t$ . It is assumed that  $v$  is the list of genuine lines, i.e. that empty lines and comment lines have been removed.

```

[[lgc-file2lines ( f ) ≡ lgc-file2lines1 ( f , T , T )]]]

```

Convert the list  $f$  of singleton strings into a list of lines, removing empty lines and comment lines.

```

[[lgc-file2lines1 ( f , l , b ) ≡
if f ∈ A then reverse ( lgc-add-line-to-lines ( b , l ) ) else
let c::f = f in
if c = LF or c = CR then
lgc-file2lines1 ( f , lgc-add-line-to-lines ( b , l ) , T ) else
lgc-file2lines1 ( f , l , c::b )]]]

```

Convert the list  $f$  of singleton strings into a list of lines. Lines are accumulated in  $l$  in reverse order. Characters of each line are buffered in  $b$  in reverse order.

```
[lgc-add-line-to-lines ( b , l ) ≡  
  let b = reverse ( b ) in  
  if b ∈ A then l else  
  if bh = '#' then l else  
  vt2vector ( b ) :: l]
```

Add the line  $b$  to the list  $l$  of lines unless  $b$  is empty or starts with a hash-mark.

```
[lgc-process-file ( f , t ) ≡  
  let l = lgc-file2lines ( f ) in  
  lgc-process-lines ( l , t )]
```

Process the configuration file  $f$  and add parameter values to the array  $t$ .

```
[lgc-process-parameters ( s ) ≡  
  let ⟨⟨T, T :: a, e, c, d⟩⟩ = s['init'] in  
  let t = lgc-default-options in  
  let t = lgc-process-lines ( d , t ) in  
  let u = t['siteconfig'] in  
  let t = lgc-process-file ( s['siteconfig'] , t ) in  
  let v = t['userconfig'] in  
  let t = lgc-process-file ( s['userconfig'] , t ) in  
  let t = t['siteconfig' → u] ['userconfig' → v] in  
  let t = lgc-process-env ( e , t ) in  
  let t = lgc-process-argv ( a , t ) in  
  t]
```

Process parameters from all sources in  $s$ .

## 1.11 Query processing

Some invocations of the lgc compiler queries information rather than asking for compilation. The query options are **help**, **help2**, **help3**, **version**, **license**, **options**, and **option**. The three first give preset responses, the fourth lists all options, and the last lists the value of a particular option.

```
[lgc-help ≡  
,
```

```
Logiweb compiler (lgc)  
Usage: lgc [option]... [source]  
--siteconfig or -s   Location of site configuration file  
--userconfig or -u   Location of user configuration file  
--path      or -p    Path for searching pages by reference
```

```

--namepath or -n Path for searching pages by name
--rendering or -r Location of rendering output
--link or -l Location of links to rendering output
--verbose or -v Set verbosity
--option or -o Print given option and exit
--help or -h Print present message and exit
--help2 or -H Print help on further options

```

Examples:

```

--path=x or -p x Set path to singleton x
--path+x or +p x Add x in front of path
--path++x or ++p x Add x at end of path
--path Set path to the empty list

```

']

[lgc-help2 **••**  
,

```

Logiweb compiler (lgc)
Usage: lgc [option]... [source]
--help or -h Print help on frequently used options
--help2 or -H Print present message
--help3 Print help on config file options
--version Print version
--license Print license
--options Print all options
--source E.g. 'lgc --source=X' is like 'lgc X'

```

']

[lgc-help3 **••**  
,

```

Logiweb compiler (lgc)
Usage: lgc [option]... [source]
--help or -h Print help on frequently used options
--help2 or -H Print help on further options
--help3 Print present message and exit

```

The following options typically occur (without hyphens) in configuration files or at the end of scripts.

```

--leap Location of leap seconds
--newline Host newline (CR, LF, CRLF, or LFCR)
--script Script headline (e.g. #!/usr/bin/lgwam script)

```

']

```
[lgc-version ≐
  ‘Logiweb compiler (lgc) version ’::‘x.x.x
’]
```

```
[lgc-license ≐
  ‘Logiweb, a system for electronic distribution of mathematics
  Copyright (C) 2004-2009 Klaus Grue
```

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed IN the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 021111307 USA

Contact: Klaus Grue, DIKU, Universitetsparken 1, DK2100 Copenhagen Denmark, grue@diku.dk, <http://logiweb.eu/>, <http://www.diku.dk/~grue/>

Logiweb is a system for distribution of mathematical definitions, lemmas, and proofs. For more on Logiweb, consult <http://logiweb.eu/>.

```
[lgc-options ( t ) ≐
  ⟨write request ( reverse ( lgc-options1 ( t , T ) ) )⟩]
  Print all options in the array t (in peculiar order).
```

```
[lgc-options1 ( t , r ) ≐
  if t ∈ A then r else
  if th ∈ Z then lgc-options2 ( tt , LF::th::r ) else
  lgc-options1 ( th , lgc-options1 ( tt , r ) )]
  Add options in t to the list r.
```

```
[lgc-options2 ( t , r ) ≐
  if t ∈ A then r else
  lgc-options2 ( tt , LF::th::‘ ’::r )]
```

Add list  $t$  of values to  $r$ .

```
[lgc-option ( s ) ≡
  if s ∈ A then ⊤ else
  writeln request ( sh ) :: lgc-option ( st )]
```

```
[lgc-process-query ( s ) ≡
  let t = lgc-process-parameters ( s ) in
  let s = s[‘parameters’→t] in
  if t[‘help’]h ≠ ‘no’ then ⟨write request ( lgc-help )⟩ else
  if t[‘help2’]h ≠ ‘no’ then ⟨write request ( lgc-help2 )⟩ else
  if t[‘help3’]h ≠ ‘no’ then ⟨write request ( lgc-help3 )⟩ else
  if t[‘version’]h ≠ ‘no’ then ⟨write request ( lgc-version )⟩ else
  if t[‘license’]h ≠ ‘no’ then ⟨write request ( lgc-license )⟩ else
  if t[‘options’]h ≠ ‘no’ then lgc-options ( t ) else
  let o = t[‘option’]h in
  if o ≠ ⊤ and o ≠ ‘ ’ then lgc-option ( t[o] ) else
  let v = t[‘verbose’]h in
  let e :: v = lgc-atoi ( vector2vector* ( v ) , ⊤ )o in
  if e then ⟨writeln request ( ‘Invalid verbosity’ )⟩ else
  let s = s[‘verbose’→v] in
  let e :: s = lgc-process-leap ( s )o in
  if e then ⟨writeln request ( s )⟩ else
  let s = s[‘time’→lgc-unix2lgt ( s[‘time’] , s )] in
  lgc-lex-1 ( s )]
```

## 2 Time functions

### 2.1 Time schemes

Logiweb uses the following time schemes

TAI International atomic time  
UTC Universal coordinated time  
MJD Modified Julian day  
GRD Gregorian date  
LGT Logiweb time

### 2.2 TAI

TAI (International atomic time) is a ‘paper’ clock in the sense that it is a computed average of lots of real, atomic clocks located all over the world.

TAI counts seconds, minutes, and hours as regularly as possible. Each TAI day has 24 TAI hours, each TAI hour has 60 TAI minutes, and each TAI minute has 60 TAI seconds. Each TAI second is, as closely as possible, one SI second.

An SI second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom.

TAI time is independent of the rotation of planet Earth.

In Logiweb, TAI hour *hh*, minute *mm*, and second *ss* is written TAI:hh:mm:ss. We have  $00 \leq hh \leq 23$ ,  $00 \leq mm < 59$ , and  $00 \leq ss \leq 59$ . Occasionally, we shall use TAI:24:00:00 to denote TAI:00:00:00 on the following day. Decimal fractions of a second are written after a dot as in TAI:12:23:34.456 which denotes 0.456 seconds past TAI:12:23:34. The notation is compatible with ISO 8601 except that we prepend “TAI:” to emphasize the use of International Atomic Time.

## 2.3 UTC

UTC is a combination of TAI and yet another time scale named UT1.

UT1 is a measure of the rotation angle of planet Earth relative to the direction from the Earth to the Sun. Each UT1 day has 24 UT1 hours, each UT1 hour has 60 UT1 minutes, and each UT1 minute has 60 UT1 seconds. It is noon in UT1 when Greenwich is under the Sun. At the time of writing (February 2009), UT1 is around 34 seconds behind TAI. In 1972, UT1 was 10 seconds behind TAI.

As mentioned, UTC is a combination of TAI and UT1. UTC equals TAI plus a politically decided offset. This UTC offset indicates how much UTC lacks behind TAI. At the time of writing, the UTC offset is 34 seconds indicating that UTC is 34 seconds behind TAI.

At any time, the UTC offset is an integral number of seconds, but the UTC offset may be incremented or decremented by decree from the International Earth Rotation Service (IERS). Hence, UTC depends on TAI and IERS, but IERS has the intension to keep the difference between UTC and UT1 below 0.9 seconds, so UTC indirectly depends on UT1.

UTC makes a leap whenever IERS increments or decrements the UTC offset. Such leaps are implemented by irregular UTC minutes.

A regular UTC minute has 60 UTC seconds. An irregular one either has 59 or 61. Apart from that UTC counts like TAI and UT1: days have 24 hours and hours have 60 minutes.

Whenever IERS increments (decrements) the UTC offset, the last minute of the last hour of a particular UTC day has 61 (59) seconds. IERS intends to place irregular seconds at the end of June 30 and December 31 when necessary and intends to announce the leaps in advance.

In Logiweb, UTC hour *hh*, minute *mm*, and second *ss* is written UTC:hh:mm:ss. We have  $00 \leq hh \leq 23$ ,  $00 \leq mm < 59$ , and  $00 \leq ss \leq 60$ . Occasionally, we shall use UTC:24:00:00 to denote UTC:00:00:00 on the following day. Decimal fractions of a second are written after a dot as in UTC:12:23:34.456 which denotes 0.456 seconds past UTC:12:23:34. The notation is compatible with ISO 8601 except that we prepend “UTC:” to emphasize the use of Universal Coordinated Time.

At the time of writing, IERS has never decremented the UTC offset, but has incremented the UTC offset at the end of the following days:

GRD-1972-06-30  
GRD-1972-12-31  
GRD-1973-12-31  
GRD-1974-12-31  
GRD-1975-12-31  
GRD-1976-12-31  
GRD-1977-12-31  
GRD-1978-12-31  
GRD-1979-12-31  
GRD-1981-06-30  
GRD-1982-06-30  
GRD-1983-06-30  
GRD-1985-06-30  
GRD-1987-12-31  
GRD-1989-12-31  
GRD-1990-12-31  
GRD-1992-06-30  
GRD-1993-06-30  
GRD-1994-06-30  
GRD-1995-12-31  
GRD-1997-06-30  
GRD-1998-12-31  
GRD-2005-12-31  
GRD-2008-12-31

Before GRD-1972-06-03, the UTC offset was 10 seconds.

## 2.4 MJD

MJD (Modified Julian Day) is a scheme for counting days in a completely regular fashion. Each day is simply expressed by the number of days since a particular day.

MJD is a regular and reliable day count used by astronomers. Furthermore, it is politically correct in the sense that, even though Julius Caesar was quite controversial in his own time, few people today are offended by a time scale named after him.

MJD is based on yet another time scale named JD (Julian Day). JD expresses the number of days since noon, January 1, year -4712 (year 4713 BC), in the Julian calendar.

In ancient times, a day was measured from noon to noon, so people actually counted nights instead of days (as a reminiscence, a period of 14 days is still called a fortnight in the English tongue). Today, we prefer to step our day counters when the sun is on the other side of the planet, which is of course difficult to observe, but which possesses little problem for modern technology.

To get a day count based on JD which steps at midnight, the Modified Julian Day (MJD) is offset from JD by 2400000.5 days. In consequence, MJD counts the number of days since GRD-1858-11-17.

When we combine MJD with UTC, then MJD steps at UTC:00:00:00. When we combine MJD with TAI, then MJD steps at TAI:00:00:00. Hence, MJD/UTC and MJD/TAI are two different day counts, but at the time of writing they merely differ by 34 seconds.

In Logiweb, MJD day  $d$  is written MJD- $d$ . As an example, GRD-1858-11-17 equals MJD-0 and MJD-51544 equals GRD-2000-01-01. The day before MJD-0 is named MJD-1 (i.e. Modified Julian Day hyphen minus one). The notation follows ISO 8601 in using a hyphen in connection with day counting, but is otherwise completely unrelated.

Combinations of day and second counting schemes are glued together with a dot. As an example, 0.456 seconds past TAI:12:23:34 on MJD-51544 is written MJD-51544.TAI:12:23:34.456. This follows ISO-8601 in putting the day before the second but does not follow the suggestion of ISO-8601 to separate day and second by a capital “T”.

## 2.5 GRD

GRD (Gregorian Date) is a scheme for counting days in a fashion so complicated that it has taken millennia to screw it up. Furthermore, GRD is “politically incorrect” in that it counts days, not after “Jesus”, but after “The Lord” (Anno Domini), which is not completely neutral.

But GRD is widespread, and hence we use it in the Human-Computer-Interfaces of Logiweb. In Logiweb itself, GRD has no place.

In GRD, day 0 of a year is named “January 1”, and day 100 is named April 11 (except if the year is divisible by 4, in which case it is named April 10 (except if the year is divisible by 100, in which case it is named April 11 (except if the year is divisible by 400, in which case it is named April 10))).

In Logiweb, Gregorian year  $Y$ , month  $MM$ , and day  $DD$  is written GRD- $Y$ - $MM$ - $DD$ . We have  $01 \leq MM \leq 12$  and  $01 \leq DD \leq 31$ . The notation is compatible with ISO 8601 except for the following: (1) We prepend “GRD-” to emphasize that we label days like the Gregorian calendar does (GRD for GREGorian Date). (2) We allow the year to have more than four digits after year 9999 and to have less than four digits before year 1000. (3) We allow the year to be zero and negative. As examples, GRD-0-01-01 and GRD-5-01-01 are January 1 on year 1 BC and 6 BC, respectively.

When we combine GRD with UTC, then GRD steps at UTC:00:00:00. When we combine GRD with TAI, then GRD steps at TAI:00:00:00. Hence, GRD/UTC and GRD/TAI are two different day counts, but at the time of writing they merely differ by 34 seconds.

Combinations of day and second counting schemes are glued together with a dot. As an example, 0.456 seconds past UTC:12:23:34 on GRD-2000-01-01 is written GRD-2000-01-01.UTC:12:23:34.456. This follows ISO-8601 in putting

the day before the second but does not follow the suggestion of ISO-8601 to separate day and second by a capital “T”.

## 2.6 LGT

LGT (Logiweb time) is the number of TAI seconds since MJD-0.TAI:00:00:00.

Logiweb time is expressed on the form  $M \cdot 10^{-E}$  where M is an integer and E is a cardinal (i.e. a non-negative integer). In Logiweb, M is always non-negative, so one could as well say that M is a cardinal.

Logiweb time  $M \cdot 10^{-E}$  is written LGW-Me-E. As an example,

LGW-1083564821686603e-6

equals

GRD.2004-05-03.UTC:06:13:41.686603.

The e-E may be replaced by the following decadic suffixes:

Exponent	Suffix	Name
e-0	U	unit
e-3	m	milli
e-6	u	micro
e-6	$\mu$	micro
e-9	n	nano
e-12	p	pico
e-15	f	femto
e-18	a	atto
e-21	z	zepto
e-24	y	yocto

In a Logiweb time like LGW-1083564821686603e-6 one should not replace the small e by a capital one as that may cause confusion with the decadic suffix E (Exa) which stands for  $10^{15}$ .

By the way note the following: Logiweb is a computational system intended for mathematics. In physics, one uses decadic prefixes that glue in front of physical units. In computing systems it is better to use decadic suffixes that glue behind numbers. When needed, Logiweb uses the SI units meter, kilogram, second, etc., and derived units. As an example, font sizes are measured in meters. A font size of twelve typographic points is 4218u, and a printer with a resolution of 600 dots per 0.0254 meters has a distance between pixels of 42.333u. An area of 1m by 1m (one milli meter by one milli meter) is 1m<sup>2</sup> (one square milli) or 1u (one micro) or 1e-6 measured in the derived SI unit of square meters. A weight of 1m is one gram (one milli kilogram). A weight of 1um is one microgram (one micro milli kilogram). This is almost but not completely different from the use of decadic prefixes in the SI system.

## 2.7 GUTC

When presenting Logiweb time to a user, we use GRD/UTC which we shall refer to as GUTC. This section describes GUTC in more detail than the individual sections on GRD and UTC.

GUTC is irregular compared to Logiweb time in that it occasionally includes leap seconds and, furthermore, it counts days in a rather complicated (Gregorian) manner, which includes leap days.

GUTC is built up from the following cycles:

**GUTC second.** The length of a GUTC second is one SI second (which equals one TAI and one UTC second). Each GUTC second starts at the 'tick' of the TAI 'paper' clock. As an example, the duration from UTC:00:00:00 to UTC:00:00:01 on GRD-2000-03-01 (March 1, year 2000) is a GUTC second.

**GUTC minute.** Regular GUTC minutes consist of 60 GUTC seconds. Irregular GUTC minutes consist of 61 or 59 GUTC seconds. As an example, the duration from UTC:00:00:00 to UTC:00.01.00 on GRD-2000-03-01 is a regular GUTC minute.

**GUTC hour.** Regular GUTC hours consist of 60 regular GUTC minutes. Irregular GUTC hours consist of 59 regular GUTC minutes followed by one irregular GUTC minute. As an example, the duration from UTC:00:00:00 to UTC:01.00.00 on GRD-2000-03-01 is a regular GUTC hour.

**GUTC day.** Regular GUTC days consist of 24 regular GUTC hours. Irregular GUTC days consist of 23 regular GUTC hours followed by one irregular GUTC hour. As an example, the duration from GRD-2000-03-01.UTC:00:00:00 to GRD-2000-03-02.UTC:00.00.00 is a regular GUTC day.

**Long GUTC month.** A long GUTC month consists of 31 GUTC days. As an example, the duration from GRD-2000-03-01.UTC:00:00:00 to GRD-2000-04-01.UTC:00.00.00 (i.e. March) is a long GUTC month.

**Short GUTC month.** A short GUTC month consists of 30 GUTC days. As an example, the duration from GRD-2000-04-01.UTC:00:00:00 to GRD-2000-05-01.UTC:00.00.00 (i.e. April) is a short GUTC month.

**GUTC dimester.** A GUTC dimester (dimester = two months, compare trimester = tres menses = three months and semester = sex menses = six months) consists of a long GUTC month followed by a short one. As an example, the duration from March to April (inclusive) is a GUTC dimester.

**GUTC quimester.** A GUTC quimester (quimester = five months) consists of a long, a short, a long, a short, and a long GUTC month. In other words, a quimester consists of two regular dimesters followed by an irregular one that ends abruptly at the end of the quimester. As an example, the duration from March to July (inclusive) is a GUTC quimester. The duration from August to December is another quimester.

**GUTC Roman year.** A GUTC Roman year is the duration from March 1, inclusive, to the following March 1, exclusive. A regular Roman year has 365 GUTC days; an irregular one has one more. As an example, the period from GRD-2000-03-01.UTC:00:00:00 to GRD-2001-02-28.UTC:24:00:00 is GUTC Roman year 2000, which is regular. In contrast, the Gregorian year 2000 is a leap

year and, hence, irregular. The difference arises because the Gregorian and Roman years have newyear before and after the leap day, respectively.

A GUTC Roman year consists of three quimesters, the third of which ends abruptly at the end of the year. As a consequence of the conventions mentioned until now, the last month of a regular GUTC Roman year (February) gets 28 GUTC days, and all the other months gets 30 and 31 GUTC days in the pattern prescribed by the Gregorian calender.

**GUTC olympiad.** A regular GUTC olympiad consists of three regular GUTC years followed by an irregular one. An irregular GUTC olympiad consists of four regular GUTC years. As an example, the period from March 1, 2000 to February 29, 2004 is a regular GUTC olympiad.

**GUTC century.** A regular GUTC century consists of 24 regular GUTC olympiads followed by an irregular one. An irregular GUTC olympiad consists of 25 regular GUTC olympiads. As an example, March 1, 2000 to February 28, 2100 is a regular GUTC century.

**GUTC Gregorian cycle.** A GUTC Gregorian cycle consists of three regular GUTC centuries followed by an irregular one.

The rules above allow to convert from LGT to GUTC and back provided one knows the location of leap seconds. A historical 'Roman year' just had 10 months, starting around vernal equinox. 10 month after vernal equinox, the romans stopped counting days and just waited for the next vernal equinox. The dimester/quimester structure described above is accidental. For the sake of Roman political correctness, the month of August (named after Augustus) was extended to 31 days to make it as long as July (named af Julius).

## 2.8 Unix time

Unix time counts seconds since GRD-1970-01-01.UTC:00:00:00. Each Unix day has 24 Unix hours, each Unix hour has 60 Unix minutes, and each Unix minute has 60 Unix seconds. Most Unix seconds are approximately equal to one SI second.

Each Unix computer maintains its own Unix time and each Unix computer tries to keep its Unix time in sync with UTC. When Unix time on a computer lacks behind UTC then the Unix system on that computer makes the Unix clock on that computer run faster until Unix time on that computer catches up with UTC. Likewise, the Unix system slows down the Unix clock when needed.

Whenever UTC makes a leap, Unix time makes a stumble. As an example, Unix time may stumble as follows: Whenever UTC makes a positive leap, Unix time counts at half speed for two seconds. In that way Unix time keeps in sync with UTC in the long run.

As can be seen, a Unix second is not always an SI second. Exactly how Unix time stumbles depends on the version of Unix used.

One can get a poor mans TAI from Unix time by stumbling backwards. That is what the Logiweb compiler does. When converting Unix time to poor mans TAI we assume that each Unix second corresponds to one TAI second except when a leap second occurs. Whenever UTC makes a positive leap, we

assume that Unix time counts at half speed from UTC:23:59:59 to UTC:23:59:61 (UTC:23:59:61 equals UTC:00:00:00 on the next day). Whenever UTC makes a negative leap, we assume that Unix time counts at double speed from UTC:23:59:58 to UTC:23:59:59 (UTC:23:59:59 equals UTC:00:00:00 on the next day).

## 2.9 The purpose of time stamps

One purpose of using time stamps in Logiweb is as follows: A Logiweb reference includes a RIPEMD code. If the reference contained nothing else, there would be a theoretical limit of  $2^{160}$  Logiweb pages. Since Logiweb is supposed, among other, to support mathematical logic, it is convenient to have no upper limit on the number of pages, not even a theoretical one. Otherwise, one could fear pathological metatheorems stating that certain theorems fail because their proof would require more than  $2^{160}$  Logiweb pages to prove. As a countermeasure, Logiweb references include a timestamp. To ensure that there is not even an upper limit on the number of pages that can be published per second, the timestamp comprises an exponent and a mantissa, allowing time stamps to have arbitrary resolution.

Another reason for including a time stamp in Logiweb references is thus: RIPEMD ensures that the probability of generating two Logiweb pages with the same reference is negligible. However, adding a timestamp makes it possible to recover even in the theoretical situation where two pages get the same RIPEMD code anyway: One may simply resubmit both pages so that they get new time stamps.

A third reason for including a time stamp is thus: Logiweb pages are allowed to reference each other, but the pages and references are required to form a directed, acyclic graph. That can be enforced by a rule stating that a page is only allowed to reference pages which are older according to their time stamp. That is not used in the present version of Logiweb because RIPEMD is trusted to ensure acyclicity: If one tries to produce two Logiweb pages which reference each other, then the RIPEMD codes of each page will depend on the RIPEMD code of the other, making it infeasible to construct those codes.

Accidentally, the time stamps turned out to be quite convenient when rendering pages using  $\text{\LaTeX}$ : Some  $\text{\TeX}$  and  $\text{\LaTeX}$  styles need the date and time of submission to generate a front page, and that date and time may conveniently be taken from the time stamp of the page.

## 2.10 Parameter names

In the time conversion functions we use variable names thus:

$S$	state
$g$	Gregorian cycle (400 year period)
$c$	Century (100 year period)
$o$	Olympiad (4 year period)
$Y$	Year
$q$	Quimester (5 month period)
$d$	Dimester (2 month period)
$M$	Month
$D$	Day
$h$	Hour
$m$	minute
$s$	second
$f$	fraction
$e$	exponent
$G$	Gregorian date $\langle Y, M, D \rangle$

## 2.11 Constants

[lgc-seconds-per-minute  $\stackrel{\bullet\bullet}{\equiv}$  60]

[lgc-seconds-per-day  $\stackrel{\bullet\bullet}{\equiv}$  24 · 60 · 60]

[lgc-minutes-per-hour  $\stackrel{\bullet\bullet}{\equiv}$  60]

[lgc-hours-per-day  $\stackrel{\bullet\bullet}{\equiv}$  24]

[lgc-days-per-month  $\stackrel{\bullet\bullet}{\equiv}$  31]

[lgc-days-per-dimester  $\stackrel{\bullet\bullet}{\equiv}$  31 + 30]

[lgc-days-per-quimester  $\stackrel{\bullet\bullet}{\equiv}$  31 + 30 + 31 + 30 + 31]

[lgc-days-per-year  $\stackrel{\bullet\bullet}{\equiv}$  365]

[lgc-days-per-olympiad  $\stackrel{\bullet\bullet}{\equiv}$  4 · lgc-days-per-year + 1]

[lgc-days-per-century  $\stackrel{\bullet\bullet}{\equiv}$  25 · lgc-days-per-olympiad - 1]

[lgc-days-per-Gregorian  $\stackrel{\bullet\bullet}{\equiv}$  4 · lgc-days-per-century + 1]

[lgc-months-per-dimester  $\stackrel{\bullet\bullet}{\equiv}$  2]

[lgc-months-per-quimester  $\stackrel{\bullet\bullet}{\equiv}$  5]

[lgc-months-per-year  $\stackrel{\bullet\bullet}{\equiv}$  12]

[lgc-month-of-march  $\stackrel{\bullet\bullet}{\equiv}$  3]

March is month number three in the Gregorian calendar.

[lgc-years-per-olympiad  $\stackrel{\bullet\bullet}{\equiv}$  4]

[lgc-years-per-century  $\equiv$  100]

[lgc-years-per-Gregorian  $\equiv$  400]

This is the length of the Gregorian cycle.

[lgc-grd-of-mjd0  $\equiv$  (1858, 11, 17)]

## 2.12 Conversion from GRD to MJD

[lgc-grd2day (  $G$  )  $\equiv$

let  $\langle Y, M, D \rangle = G$  in

let  $M = \text{lgc-months-per-year} \cdot Y + M - \text{lgc-month-of-march}$  in

let  $Y :: M = \text{floor} ( M , \text{lgc-months-per-year} )$  in

let  $g = Y \text{ div } \text{lgc-years-per-Gregorian}$  in

let  $c = Y \text{ div } \text{lgc-years-per-century}$  in

let  $o = Y \text{ div } \text{lgc-years-per-olympiad}$  in

let  $q :: M' = \text{floor} ( M , \text{lgc-months-per-quimester} )$  in

let  $d = M' \text{ div } \text{lgc-months-per-dimester}$  in

$D - 1 + M \cdot \text{lgc-days-per-month} + Y \cdot \text{lgc-days-per-year} - d - q \cdot 2 + o - c + g$ ]

Compute the number of days since GRD-0-03-01 as follows:

Destruct the Gregorian date  $G$  into year  $Y$ , month  $M$ , and day  $D$ . Then compute the month count  $M$  since March, Gregorian year zero (recall that year zero is the one before year one, i.e. year one BC).

Then compute the Roman year  $Y$  and month  $M$ . We count from zero so that March is month zero. The Romans counted March as month one. So October is month eight according to the Romans (Octo=8) but we count it as month seven.

After that, compute the day count  $D$  since GRD-0-03-01 as a linear combination of day, month, and year, and adjust for variations in the length of months and years.

[lgc-day-of-mjd0  $\equiv$  lgc-grd2day ( lgc-grd-of-mjd0 )]

[lgc-grd2mjd (  $G$  )  $\equiv$  lgc-grd2day (  $G$  ) - lgc-day-of-mjd0]

## 2.13 Conversion from MJD to GRD

[lgc-limited-floor (  $x$  ,  $y$  ,  $l$  )  $\equiv$

let  $R = \text{floor} ( x , y )$  in

if  $R^h < l$  then  $R$  else  $l :: x - l \cdot y$ ]

Compute  $q :: r$  such that  $x = q \cdot y + r$ . The quotient  $q$  is the one computed by  $x \text{ div } y$  except that it cannot exceed  $l$ .

[lgc-mjd-of-grd-0-03-01  $\equiv$  lgc-grd2mjd ( (0, 3, 1) )]

```

[lgc-mjd2grd ( D ) ≐
  let D = D - lgc-mjd-of-grd-0-03-01 in
  let g :: D = floor ( D , lgc-days-per-Gregorian ) in
  let c :: D = lgc-limited-floor ( D , lgc-days-per-century , 3 ) in
  let o :: D = floor ( D , lgc-days-per-olympiad ) in
  let Y :: D = lgc-limited-floor ( D , lgc-days-per-year , 3 ) in
  let q :: D = floor ( D , lgc-days-per-quimester ) in
  let d :: D = floor ( D , lgc-days-per-dimester ) in
  let M :: D = floor ( D , lgc-days-per-month ) in
  let M = M + d · lgc-months-per-dimester in
  let M = M + q · lgc-months-per-quimester in
  let Y = Y + o · lgc-years-per-olympiad in
  let Y = Y + c · lgc-years-per-century in
  let Y = Y + g · lgc-years-per-Gregorian in
  let M = M + Y · lgc-months-per-year in
  let Y :: M = floor ( M+lgc-month-of-march-1 , lgc-months-per-year
  ) in
  ⟨ Y , M + 1 , D + 1 ⟩]

```

Convert the Modified Julian Day  $D$  to a Gregorian date  $\langle Y, M, D \rangle$  as follows:

First compute the number of days  $D$  since GRD-0-03-01 (March 1, Gregorian year zero). Then convert  $D$  into a number of Gregorian cycles  $g$ , centuries  $c$ , olympiads  $o$ , years  $Y$ , quimesters  $q$ , dimesters  $d$ , months  $M$ , and days  $D$ . Then combine  $g$ ,  $c$ ,  $o$ , and  $Y$  into the number of years  $Y$  since GRD-0-03-01, and convert  $q$ ,  $d$ , and  $M$  into the number of months that have elapsed on top of those years. Then combine  $M$  and  $Y$  into the number of months  $M$  since GRD-0-03-01. Then move newyear to GRD-0-01-01 and convert back to year  $Y$  and month  $M$ . Finally construct the Gregorian date, taking into account that month and day count from one.

## 2.14 GRD Parsing functions

```

[lgc-parse-prefix ( p , a ) ≐
  if p ∈ A then a else
  if ph ≠ ah then • else
  lgc-parse-prefix ( pt , at )]

[lgc-prefix-grd ≐ vt2vector* ( 'GRD-' )]

[lgc-prefix-hyphen ≐ vt2vector* ( '-' )]

[lgc-parse-leap ( a ) ≐
  let a = lgc-parse-prefix ( lgc-prefix-grd , a ) in
  let Y :: a = lgc-parse-int ( a ) in
  let a = lgc-parse-prefix ( lgc-prefix-hyphen , a ) in

```

```

let  $M :: a = \text{lgc-parse-int} ( a )$  in
let  $a = \text{lgc-prefix} ( \text{lgc-prefix-hyphen} , a )$  in
let  $D :: a = \text{lgc-parse-int} ( a )$  in
let  $D = \text{lgc-grd2mjd} ( \langle Y, M, D \rangle )$  in
let  $c :: a = a$  in
let  $l :: a = \text{lgc-parse-int} ( a )$  in
if  $a \in \mathbf{P}$  then  $\bullet$  else
if  $c = \text{'+'}$  then  $D :: l$  else
if  $c = \text{'-'}$  then  $D :: -l$  else  $\bullet$  ]

```

```

[lgc-convert-leap (  $L$  )  $\bullet\bullet$ 
if  $L \in \mathbf{A}$  then  $\mathbf{T}$  else
let  $a :: L = L$  in
let  $e :: v = \text{lgc-parse-leap} ( \text{vt2vector*} ( a ) )^\circ$  in
if  $e$  then  $\text{'Invalid leap: ' :: } a$   $\bullet$  else
 $v :: \text{lgc-convert-leap} ( L )$  ]

```

Convert the list  $L$  of leap seconds on external form to a list of leap seconds on form  $\langle D :: U', \dots \rangle$  where  $D$  is the day in MJD and  $U'$  is the change in UTC lack (the amount UTC lacks behind TAI). The change  $U'$  in UTC lack is +1 for a positive leap, -1 for a negative one.

```

[lgc-check-leap (  $L$  )  $\bullet\bullet$ 
if  $L^t \in \mathbf{A}$  then  $\mathbf{T}$  else
if  $L^{\text{hh}} \leq L^{\text{thh}}$  then
 $\text{'Leap seconds must be stated in descending order'}$   $\bullet$  else
 $\text{lgc-check-leap} ( L^t )$  ]

```

Check that the dates of leap seconds are stated in descending order.

```

[lgc-initial-leap  $\bullet\bullet$  10]

```

The UTC lack before the first announced leap second.

```

[lgc-add-leap (  $L$  )  $\bullet\bullet$ 
lgc-add-leap1 (  $\text{reverse} ( L )$  ,  $\text{lgc-initial-leap}$  ,  $\mathbf{T}$  ) ]

```

Convert the list  $L$  on form  $\langle D :: U', \dots \rangle$  into a list of form  $U :: \langle s :: U', \dots \rangle$ . The value of  $U'$  is unchanged. The value of  $U$  is the UTC lack after all the leap seconds have occurred. The value of  $s$  is the end of the day  $D$  expressed in TAI. The “end of the day” equals the beginning of the next day, i.e. the point in time where the leap ends. Note that negative leaps are instantaneous so that they begin and end on the same moment whereas positive leaps have a duration of one second.

```

[lgc-add-leap1 (  $L$  ,  $U$  ,  $r$  )  $\bullet\bullet$ 
if  $L \in \mathbf{A}$  then  $U :: r$  else
let  $( D :: U' ) :: L = L$  in
let  $U = U + U'$  in
```

**let**  $s = (D + 1) \cdot \text{lgc-seconds-per-day} + U$  **in**  
 $\text{lgc-add-leap1} ( L , U , (s :: U') :: r )$

$[\text{lgc-process-leap} ( s ) \equiv$

**let**  $L = s[\text{'parameters'}][\text{'leap'}]$  **in**  
**let**  $L = \text{lgc-convert-leap} ( L )$  **in**  
 $\text{lgc-check-leap} ( L )$  .then.  
**let**  $L = \text{lgc-add-leap} ( L )$  **in**  
 $s[\text{'leap'} \rightarrow L]$

Extract the leap parameter from the state  $s$ , convert it, and store it back into  $s[\text{'leap'}]$ .

## 2.15 Conversion from reference to Logiweb time

The  $\text{lgc-ref2lgt} ( r )$  function returns the time stamp of the Logiweb reference  $r$  expressed in Logiweb time.

$[\text{lgc-ref2lgt} ( r ) \equiv$

**let**  $r = \text{vt2vector*} ( r )$  **in**  
**let**  $v :: r = r$  **in**  
**if**  $v \neq \text{lgc-ref-version}$  **then**  $\text{lgc-panic} ( \text{'Internal error: Wrong version'} )$  **else**  
**let**  $r = \text{list-suffix} ( r , 20 )$  **in**  
**let**  $f :: r = \text{parse-card} ( r )$  **in**  
**let**  $e :: r = \text{parse-card} ( r )$  **in**  
 $\langle f, e \rangle]$

## 2.16 Conversion from Logiweb time to printed representation

$\text{lgc-lgt2vt} ( s )$  converts the Logiweb time  $s = \langle f, e \rangle$  to a human readable Logiweb time. The return value is a vector tree.

$[\text{lgc-lgt2vt} ( s ) \equiv$

**let**  $\langle f, e \rangle = s$  **in**  
**let**  $f = \text{lgc-itoa} ( f )$  **in**  
**let**  $e = \text{lgc-itoa} ( e )$  **in**  
 $\langle \text{'LGT-'}, f, \text{'e-'}, e \rangle]$

## 2.17 Conversion from Logiweb time to MJD/TAI

$\text{lgc-lgt2mjdtai} ( s )$  converts the Logiweb time  $s = \langle f, e \rangle$  to MJD and TAI on form  $\langle D, h, m, s, f, e \rangle$

$\text{lgc-lgt2mjdtai2vt} ( s )$  converts the Logiweb time  $s$  to human readable MJD and TAI. The return value is a vector tree.

The conversion is trivial because LGT expresses the number of TAI seconds since MJD-0.TAI:0:0:0 and since MJD, TAI, and LGT are all completely regular time counting schemes.

```
[lgc-lgt2mjdtai ( s ) ≐
  let ⟨f, e⟩ = s in
  let s :: f = floor ( f , exp10 ( e ) ) in
  let m :: s = floor ( s , lgc-seconds-per-minute ) in
  let h :: m = floor ( m , lgc-minutes-per-hour ) in
  let D :: h = floor ( h , lgc-hours-per-day ) in
  ⟨D, h, m, s, f, e⟩]
```

```
[lgc-lgt2mjdtai2vt ( s ) ≐
  let ⟨D, h, m, s, f, e⟩ = lgc-lgt2mjdtai ( s ) in
  let D = lgc-itoa ( D ) in
  let h = lgc-ctoa ( h , 2 ) in
  let m = lgc-ctoa ( m , 2 ) in
  let s = lgc-ctoa ( s , 2 ) in
  let f = if e = 0 then T else ‘.’ :: lgc-ctoa ( f , e ) in
  ⟨‘MJD-’, D, ‘.TAI:’, h, ‘.’, m, ‘.’, s, f⟩]
```

## 2.18 Conversion from Logiweb time to GRD/UTC

At any time, UTC is a UTC lack  $U$  behind TAI. In 1972, the UTC lack  $U$  was 10 seconds.

We represent the location of leap seconds by a list  $L$  of form  $\langle s' :: U', \dots \rangle$  where  $s'$  indicates the end time of a UTC leap expressed in LGT and  $U'$  is +1 for a positive leap and  $-1$  for a negative leap. At any time  $T$ , the UTC lack equals 10 seconds plus the values of  $U'$  for all leap seconds before  $T$ .

The `lgc-lgt2utc ( s , U , L )` function takes Logiweb second  $s$ , a leap second list  $L$ , and a UTC lack  $U$  as input. The UTC lack  $U$  must be the lack that applies after the last leap second in  $L$ .

The `lgc-lgt2utc ( s , U , L )` returns a pair  $u :: l$ . The value of  $l$  is zero in case a positive UTC leap is in progress. At UTC time  $\langle h, m, s \rangle$  of MJD  $D$ , the value of  $u$  is  $((D \cdot 24 + h) \cdot 60 + m) \cdot 60 + s - l$ . We shall refer to  $u$  and  $l$  as the UTC second and leap, respectively.

As time progresses, the UTC second  $u$  is incremented once per second except when a leap second occurs. When a negative leap occurs,  $u$  is incremented twice instead of once. When a positive leap occurs,  $u$  has the same value of two consecutive seconds. During the second of those two seconds, the UTC leap equals 1.

`lgc-lgt2grdutc2vt ( s , S )` expresses the Logiweb second  $s$  in human readable form using GRD and UTC. The return value is a vector tree. The `lgc-lgt2grdutc2vt ( s , S )` function gets the UTC lack  $U$  and leap second list  $L$  from the state  $S$  and uses `lgc-lgt2utc ( s , U , L )` to convert to UTC second  $u$  and leap  $l$ . Then it converts  $u$  into year  $Y$ , month  $M$ , day  $D$ , hour  $h$ , minute  $m$ , and second  $s$ , and finally adds  $l$  to  $s$ .

lgc-lgt2grdutc2v ( s , S ) is like lgc-lgt2grdutc2vt ( s , S ) except that it returns a vector.

```
[lgc-lgt2utc ( s , U , L ) ≡
  if L ∈ A then s - U :: 0 else
  let (s' :: U') :: L = L in
  if s' ≤ s then s - U :: 0 else
  if s' ≤ s + U' then s - U :: U' else
  lgc-lgt2utc ( s , U - U' , L )]
```

```
[lgc-lgt2grdutc ( s , S ) ≡
  let U :: L = S[‘leap’] in
  let ⟨f, e⟩ = s in
  let s :: f = floor ( f , exp10 ( e ) ) in
  let s :: l = lgc-lgt2utc ( s , U , L ) in
  let m :: s = floor ( s , lgc-seconds-per-minute ) in
  let h :: m = floor ( m , lgc-minutes-per-hour ) in
  let D :: h = floor ( h , lgc-hours-per-day ) in
  let ⟨Y, M, D⟩ = lgc-mjd2grd ( D ) in
  ⟨Y, M, D, h, m, s + l, f, e⟩]
```

```
[lgc-lgt2grdutc2vt ( s , S ) ≡
  let ⟨Y, M, D, h, m, s, f, e⟩ = lgc-lgt2grdutc ( s , S ) in
  let Y = lgc-itoa ( Y ) in
  let M = lgc-ctoa ( M , 2 ) in
  let D = lgc-ctoa ( D , 2 ) in
  let h = lgc-ctoa ( h , 2 ) in
  let m = lgc-ctoa ( m , 2 ) in
  let s = lgc-ctoa ( s , 2 ) in
  let f = if e = 0 then T else ‘.’ :: lgc-ctoa ( f , e ) in
  ⟨‘GRD-’, Y, ‘-’, M, ‘-’, D, ‘.UTC:’, h, ‘:’, m, ‘:’, s, f⟩]
```

```
[lgc-lgt2grdutc2v ( s , S ) ≡
  vt2vector ( lgc-lgt2grdutc2vt ( s , S ) )]
```

## 2.19 Conversion from Unix time to Logiweb time

As mentioned, the Logiweb compiler generates a poor mans TAI from Unix time. More specifically, the function below converts from Unix time to Logiweb time:

```
[lgc-unix2lgt ( s , S ) ≡
  let U :: L = S[‘leap’] in
  let ⟨f, e⟩ = s in
  let E = exp10 ( e ) in
  let s :: f = floor ( f , E ) in
  let s = s + lgc-lgt-of-unix + U in
  lgc-unix2lgt1 ( s , f , E , e , L )]
```

[lgc-lgt-of-unix  $\stackrel{\bullet\bullet}{\equiv}$  lgc-grd2mjd (  $\langle 1970, 1, 1 \rangle$  ) · lgc-seconds-per-day]

lgc-lgt-of-unix indicates the beginning of the TAI day containing the Unix epoch. More precisely, lgc-lgt-of-unix indicates 10 seconds before the Unix epoch since the UTC lack was 10 seconds in 1970.

[lgc-unix2lgt1 (  $s, f, E, e, L$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $L \in \mathbf{A}$  **then**  $\langle s \cdot E + f, e \rangle$  **else**  
**let**  $(s' :: U') :: L = L$  **in**  
**let**  $d = s - s'$  **in**  
**if**  $0 \leq d$  **then**  $\langle s \cdot E + f, e \rangle$  **else**  
**if**  $0 \leq d - 2 \cdot U'$  **then**  $\langle (10 \cdot s' + 5 \cdot d) \cdot E + 5 \cdot f, e + 1 \rangle$  **else**  
**if**  $0 \leq d + U'$  **then**  $\langle (s' + 2 \cdot d) \cdot E + 2 \cdot f, e \rangle$  **else**  
lgc-unix2lgt1 (  $s - U', f, E, e, L$  )]

### 3 Message generation

This section defines functions for generation of error messages, warnings, and progress messages.

#### 3.1 State entries

Message generation uses the following entries of the state:

- $s[\text{'continue'}]$  Set to false when error detected.
- $s[\text{'msg'}]$  Stack of  $p :: m$  pairs where  $p$  is a position in the source file and  $m$  is a message to be printed.

#### 3.2 Integer input/output

The lgc-atoi (  $a, m$  ) function converts the sequence  $a$  of singleton strings to an integer. The function throws  $m$  in case of error.

The lgc-itoa (  $i$  ) function converts the integer  $i$  to a list of singleton strings.

The lgc-ctoa (  $c, w$  ) function converts the cardinal  $c$  to a vector tree and pads with zeros in front to make the string at least  $w$  characters wide.

The lgc-parse-int (  $a$  ) function parses the integer at the beginning of the list  $a$  of singleton strings and returns  $i :: a'$  where  $i$  is the integer and  $a'$  is the unparsed part of  $a$ . lgc-parse-int (  $a$  ) throws an exception if no integer is found.

[lgc-parse-int (  $a$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $a \in \mathbf{A}$  **then**  $\bullet$  **else**  
**if**  $a^h \neq \text{'-'}$  **then** lgc-parse-int1 (  $a, 0$  ) **else**  
**if**  $a^t \in \mathbf{A}$  **then**  $\bullet$  **else**  
**let**  $n :: a = \text{lgc-parse-int1} ( a^t, 0 )$  **in**  $-n :: a$ ]  
Parse integer at beginning of the string  $a$ .

[lgc-parse-int1 (  $a$  ,  $r$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $a \in \mathbf{A}$  **then**  $r :: \top$  **else**  
**let**  $c = a^h$  **in**  
**if**  $c < '0'$  **or**  $c > '9'$  **then**  $r :: a$  **else**  
lgc-parse-int1 (  $a^t$  ,  $c - '0' + \text{Base} \cdot r$  )]  
Parse integer at beginning of the string  $a$ , accumulating the result in  $r$ .

[lgc-atoi (  $a$  ,  $m$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $e :: n :: a = \text{lgc-parse-int} ( a )^\circ$  **in**  
**if**  $e$  **or**  $a \neq \top$  **then**  $m^\bullet$  **else**  $n$ ]  
Convert the singleton list  $a$  to an integer.

[lgc-itoa (  $i$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $i = 0$  **then**  $\langle 0 \rangle$  **else**  
**if**  $i > 0$  **then** lgc-itoa1 (  $i$  ,  $\top$  ) **else**  $- :: \text{lgc-itoa1} ( -i , \top )$ ]  
Convert the integer  $i$  to a singleton list.

[lgc-itoa1 (  $i$  ,  $r$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $i = 0$  **then**  $r$  **else**  
**let**  $i :: m = \text{floor} ( i , \text{Base} )$  **in**  
lgc-itoa1 (  $i$  ,  $m + '0' :: r$  )]  
Convert the cardinal  $i$  to a singleton list, accumulating the result in  $r$ .

[lgc-ctoa (  $c$  ,  $w$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $c = \text{lgc-itoa1} ( c , \top )$  **in**  
repeat (  $w - \text{length} ( c )$  ,  $'0'$  )  $:: c$ ]  
Convert the cardinal  $c$  to an integer of width at least  $w$ , prepending with zeros as needed.

[lgc-ordinal-suffix (  $n$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $(n \text{ div Base}) \bmod \text{Base} = 1$  **then** th **else**  
**let**  $n = n \bmod \text{Base}$  **in**  
**if**  $n = 1$  **then** st **else**  
**if**  $n = 2$  **then** nd **else**  
**if**  $n = 3$  **then** rd **else** th]  
Return the suffix to append to a cardinal to form an ordinal (where 'cardinal' and 'ordinal' are from linguistics, not from mathematics).

[lgc-ordinal (  $n$  )  $\stackrel{\bullet\bullet}{\equiv}$   
lgc-itoa (  $n$  )  $:: \text{lgc-ordinal-suffix} ( n )$ ]  
Convert the cardinal  $n$  to an ordinal expressed as a vector tree.

### 3.3 Verbosity levels

When invoking the Logiwab compiler, the user can set the verbosity level. The default verbosity level is 3. The levels are:

- 1 Print error messages
- 2 Print warning messages
- 3 Print progress information
- 4 Print more progress information
- 5 Print debugging information

Each message has a severity level  $l$  where small values of  $l$  indicate a high severity level. Messages are only sent to the user when  $l \leq v$  where  $v$  is the verbosity.

Verbosity level zero is reserved for silent operation. The present compiler, however, does not allow suppression of error messages.

### 3.4 Message stacking

Messages are stored in  $s[\text{'msg'}]$ .

The function `lgc-add-message ( s , l , p , m )` adds the message  $m$  with severity level  $l$  in the state  $s$  provided that  $l \leq v$  where  $v$  is the verbosity.

The value of  $p$  in `lgc-add-message ( s , l , p , m )` must indicate the position in the source file which gave rise to the message. The value of  $p$  must be the position given as a byte offset. As an example,  $p = 5$  indicates the position between the fifth and sixth byte of the file. Note that  $p$  is a byte offset and not a character offset. Characters are supposed to be encoded in Logiweb Unicode UTF-8, so one character can take up several bytes. A position  $p$  can indicate a position inside a character. A negative value of  $p$  represents the end of the source file.

Once a message  $m$  is added to the state  $s$ , then  $s$  should be raised as an exception in case the message is fatal, and  $s$  should be used as the new state otherwise. The `lgc-add-message ( s , l , p , m )` adds at most ten messages to the state. When `lgc-add-message ( s , l , p , m )` adds the tenth message, it raises the resulting state as an exception.

The function `lgc-throw-message ( s , p , m )` is like `lgc-add-message ( s , l , p , m )` but always throws and always sets the severity to one.

```
[lgc-max-messages ≡ 10]
```

Maximum number of messages allowed from one run of the compiler.

```
[lgc-add-message ( s , l , p , m ) ≡  
  let s = if l > 1 then s else s[‘continue’→F] in  
  if s[‘verbose’] < l then s else
```

```

let  $M = s[\text{'msg'}]$  in
let  $M = (p :: m) :: M$  in
let  $s = s[\text{'msg'} \rightarrow M]$  in
if  $\text{length} ( M ) \geq \text{lgc-max-messages}$  then  $s^\bullet$  else  $s$ 

```

Add message  $m$  relating to position  $p$  in the source file to the state  $s$  provided the severity level  $l$  is less than or equal to the verbosity.

```

[lgc-throw-message (  $s$  ,  $p$  ,  $m$  )  $\bullet\bullet$ 
  lgc-add-message (  $s$  , 1 ,  $p$  ,  $m$  ) $\bullet$ ]

```

Like Add message but always throws and always sets the severity to one.

### 3.5 Message reporting

The `lgc-report-messages (  $s$  )` function formats messages stacked in  $s$  and convert them to a list of output events.

```

[lgc-die (  $m$  )  $\bullet\bullet$ 
  <writeln request (  $m$  ),quit request ( 1 )>]
  Die with last word  $m$ .

```

```

[lgc-report-messages (  $s$  )  $\bullet\bullet$ 
  let  $M = s[\text{'msg'}]$  in
  if  $M$  then lgc-die ( 'Unhandled exception, goodbye.' ) else
  lgc-die ( lgc-report-messages1 (  $s$  ,  $M$  , 'Goodbye.' ) )]

```

Extract messages from the state and convert them into a write request.

```

[lgc-report-messages1 (  $s$  ,  $M$  ,  $r$  )  $\bullet\bullet$ 
  if  $M \in \mathbf{A}$  then  $r$  else
  let  $(p :: m) :: M = M$  in
  if not  $p \in \mathbf{Z}$  then lgc-panic ( 'Internal error: Error position is not int' ) else
  lgc-report-messages1 (  $s$  ,  $M$  , lgc-report-messages2 (  $s$  ,  $p$  ,  $m$  ) ::  $r$  )]

```

Format the list  $M$  of messages.

```

[lgc-report-messages2 (  $s$  ,  $p$  ,  $m$  )  $\bullet\bullet$ 
  let  $f = s[\text{'source'}]$  in
  let  $p = \text{if } p < 0 \text{ then length } ( f ) \text{ else } p$  in
  let  $l :: c = \text{lgc-position} ( f , p , 1 , 1 )$  in
  let  $m_1 = \text{'Line ' } :: \text{lgc-itoa} ( l ) :: \text{' character ' } :: \text{lgc-itoa} ( c ) :: \text{' } \mathbf{in}$ 
  let  $m_2 = \text{lgc-report-message3} ( f , p )$  in
   $\text{'---' } :: \text{LF} :: m_1 :: \text{LF} :: m :: \text{LF} :: m_2 :: \text{LF}$ ]

```

Format the message  $m$  which relates to position  $p$  in the source file. Note that LF marks the end of a line regardless of the end-of-line convention of the underlying operating system.

```
[lgc-position ( f , p , L , C ) ≐
  if f ∈ A or p = 0 then L :: C else
  let c :: f = f in
  if c = LF then lgc-position-1 ( CR , f , p - 1 , L + 1 , 1 ) else
  if c = CR then lgc-position-1 ( LF , f , p - 1 , L + 1 , 1 ) else
  if c = FF then lgc-position ( f , p - 1 , L + 1 , 1 ) else
  if c = TAB then lgc-position ( f , p - 1 , L , C + 1 ) else
  if c < SP then lgc-position ( f , p - 1 , L , C ) else
  if lgc-char-start ( c ) then lgc-position ( f , p - 1 , L , C + 1 ) else
  lgc-position ( f , p - 1 , L , C )]
```

Convert position  $p$  given as a byte offset to a pair  $L :: C$  of line number  $L$  and character number  $C$  assuming Logiweb Unicode UTF-8 encoding.

```
[lgc-position-1 ( c , f , p , L , C ) ≐
  if fh = c then lgc-position ( ft , p , L , C ) else
  lgc-position ( f , p , L , C )]
```

Same as above except that character  $c$  is ignored if it occurs at the beginning of  $f$ .

```
[lgc-max-lines ≐ 3]
```

Maximum number of printed lines before and after error location.

```
[lgc-max-chars ≐ 80]
```

Maximum number of characters per line before and after error location.

```
[lgc-report-message3 ( f , p ) ≐
  let h :: t = lgc-split ( f , p , T ) in
  let h = lgc-size-limit ( h , lgc-max-lines , lgc-max-chars , T ) in
  let h = if hh = LF then ht else h in
  let t = lgc-size-limit ( t , lgc-max-lines , lgc-max-chars , T ) in
  h :: ']' :: LF :: '---' :: LF :: reverse ( tt )]
```

Split the file  $f$  into head  $h$  and tail  $t$  at position  $p$ . Then limit head and tail to at most  $\text{lgc-max-lines}$  lines of at most  $\text{lgc-max-chars}$  characters each. Then glue the head and tail together.

```
[lgc-split ( f , p , r ) ≐
  if p = 0 or f ∈ A then lgc-split1 ( r , f ) else lgc-split ( ft , p - 1 , fh :: r )]
```

Pass  $p$  bytes from  $f$  to  $r$ . Then call  $\text{lgc-split1} ( r , f )$  to move back to last character boundary.

```
[lgc-split1 ( h , t ) ≐
  if h ∈ A or t ∈ A or lgc-char-start ( th ) then h :: t else
  lgc-split1 ( ht , hh :: t )]
```

Move backwards until the tail  $t$  starts at a character boundary.

[lgc-char-start1  $\stackrel{\bullet\bullet}{\equiv}$  NULL + 128]

The value of the smallest UTF-8 byte which does not start a character.

[lgc-char-start2  $\stackrel{\bullet\bullet}{\equiv}$  NULL + 128 + 64]

One plus the value of the largest UTF-8 byte which does not start a character.

[lgc-char-start (  $c$  )  $\stackrel{\bullet\bullet}{\equiv}$   $c < \text{lgc-char-start1}$  **or**  $\text{lgc-char-start2} \leq c$ ]

True if the character  $c$  marks the start of a UTF-8 character (also true if  $c$  is illegal in UTF-8).

[lgc-size-limit (  $f$  ,  $L$  ,  $C$  ,  $r$  )  $\stackrel{\bullet\bullet}{\equiv}$

**if**  $f \in \mathbf{A}$  **or**  $L = 0$  **then**  $r$  **else**

**if**  $C = 0$  **then** lgc-size-limit (  $f$  ,  $L - 1$  , lgc-max-chars ,  $r$  ) **else**

**let**  $c :: f = f$  **in**

**let**  $r = c :: r$  **in**

**if**  $c = \text{LF}$  **then** lgc-size-limit (  $f$  ,  $L - 1$  , lgc-max-chars ,  $r$  ) **else**

**if** lgc-char-start (  $c$  ) **then** lgc-size-limit (  $f$  ,  $L$  ,  $C - 1$  ,  $r$  ) **else**

lgc-size-limit (  $f$  ,  $L$  ,  $C$  ,  $r$  )]

### 3.6 Unconditional errors

The lgc-error (  $s$  ,  $p$  ,  $a$  ) function formats the message  $a$  and converts it to a list of output events. The lgc-simple-error (  $a$  ,  $s$  ) function does the same except that it simply outputs  $a$  and exits.

[lgc-error (  $s$  ,  $p$  ,  $a$  )  $\stackrel{\bullet\bullet}{\equiv}$

**let**  $e :: s = \text{lgc-add-message}$  (  $s$  , 1 ,  $p$  ,  $a$  ) $^\circ$  **in**

lgc-report-messages (  $s$  )]

[lgc-simple-error (  $a$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$

**let**  $s = \text{lgc-progress}$  (  $a$  , 1 ,  $s$  ) **in**

lgc-do-events (  $s$  )]

### 3.7 Progress messages

The lgc-progress (  $a$  ,  $l$  ,  $s$  ) function prints the message  $a$  unless the verbosity is less than the level  $l$ .

The lgc-print (  $p$  ,  $a$  ) function prints  $a$  if  $p$  is true. The lgc-print (  $p$  ,  $a$  ) function uses print (  $a$  ) rather than constructing an event.

[lgc-progress (  $a$  ,  $l$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$

**if**  $s[\text{verbose}] < l$  **then**  $s$  **else**

lgc-push-event (  $s$  , writeln request (  $a$  ) )]

```
[lgc-print ( p , a ) ≡  
    if p then print ( a :: LF ) else T]
```

## 4 Lexical analysis

The Logiweb frontend language has a static and a dynamic syntax. The static and dynamic syntax are treated during lexical analysis and parsing, respectively. We define the static syntax in the following.

We shall refer to a sequence of two or more double quote characters as a *multiquote*. Multiquotes mark the start of an escape sequence in Logiweb source files.

We shall refer to a multiquote followed by a character or the end of the file as an *escape sequence*. Hence, an escape sequence covers both the multiquote and the first character (if any) after the multiquote.

Escape sequences that may occur in the body of a page are:

```
"   Start of string (until double quote or ".)  
""- Start of string (until double quote or ".)  
""; Start of short comment (until end of line)  
""{ Start of long comment (until "")  
""# Binary include as string (until double quote or ".)  
""$ Text include as string (until double quote or ".)  
""P Page name (until end of line or "n")  
""R Reference (until end of line or "n")  
""D Definition (until ""P, ""R, ""D or ""B)  
"". The empty string (self-terminated)  
""S Page source as a string (self-terminated)  
""N Name definitions (self-terminated)  
""C Charge defintions (self-terminated)
```

The number of double quote characters in the escape sequence closing a long comment must equal the number of double quote characters in the escape sequence opening the long comment. Apart from this, the number of double quote characters in an escape sequence is of no importance.

Some examples read:

```
"   this is a string "  
""- this is another string "  
"   this is a third string "".  
""- this is a fourth string "".  
""; this is a short comment  
""{ this is a long comment ""}  
""""{ this is another long comment """"}  
""""{ this is a strange ""}"";""{ long comment """"}  
""#/this/is/a/binary/include"  
""$/this/is/a/text/include"
```

```
"P qualifier 1 " qualifier 2 " this is a page name
"R qualifier 3 " qualifier 4 " /this/is/a/reference
"D 1.2.3 "; charge
  " + " "; production
  " - " "; another production
"B      "; body (end of definition)
". "; the empty string
"S "; Page source as a string
"N "; Name definitions
"C "; Charge definitions
```

Escapes that may occur inside a string:

```
""- No character
"! Double quote
"f Form feed
"n Line feed
"r Carriage return
"t Horizontal tab
"x Characters given in hexadecimal (until period)
```

Example:

```
"AB"-"!""x4344."!"EF".
```

The string above comprises the following eight characters:

```
AB"CD"DE
```

Escapes that may occur inside the path name part of a reference:

```
""! Double quote
```

Having an escape in a reference is useful only in the unlikely event that a file name contains a double quote character.

Escapes that may occur inside a definition:

```
"n Line feed
```

The escape sequence above allows to define more than one production on one line.

## 4.1 State entries

During lexical analysis, the compiler builds up a state  $s$  with the following contents:

- $s[\text{'source'}]$  The raw source text.

- $s$ ['sourcename'] The path of the source text (after tilde-expansion and optionally appending of .lgs).
- $s$ ['body'] The body of a page after lexical analysis. The body is a list of items of form  $c::p::S$  where  $c$  is a character,  $p$  is the position of that character in the source file, and  $S$  is the structure (if any) initiated by that structure. As an example, if  $c$  is `lgc-esc-#` then  $S$  is the name of the file to be included.
- $s$ ['page'] The page name after lexical analysis.
- $s$ ['def'] The definitions after lexical analysis.
- $s$ ['bib'] The bibliography after lexical analysis.
- $s$ ['nincludes'] Stack of pairs  $p::n$  where the values of  $n$  are names of include files and  $p$  are the positions in the source file where they occur.
- $s$ ['includes'] Array mapping names of include files to their contents.

## 4.2 Lexical analysis main functions

The `lgc-lex-1 ( s )` function requests the source file and `lgc-lex-2 ( x , s )` processes it.

```
[lgc-lex-1 ( s ) ≡
  let n = s['parameters']['source']h in
  if n then <writeln request ( 'No source file specified' )> else
  let n = lgc-tilde-expand ( n , s['parameters'] ) in
  let s = s['sourcename'→n] in
  let s = lgc-progress ( lgc-lgt2grdutc2vt ( s['time'] , s ) , 3 , s ) in
  let s = lgc-progress ( 'Reading file:'::n , 3 , s ) in
  let s = lgc-push-event ( s , fileTypeRead ( n ) ) in
  lgc-exec-events ( s , lgc-lex-2 ( x , s ) )
  Process query, if any. Else go on to compilation.
```

```
[lgc-lex-2 ( x , s ) ≡
  let <T, <T, t::f>> = x in
  if t = FileTypeRegular then lgc-lex-4 ( f , s ) else
  let n = s['sourcename']::'.lgs' in
  let s = s['sourcename'→n] in
  let s = lgc-progress ( 'Reading file:'::n , 3 , s ) in
  let s = lgc-push-event ( s , fileTypeRead ( n ) ) in
  lgc-exec-events ( s , lgc-lex-3 ( x , s ) )]
```

```
[lgc-lex-3 ( x , s ) ≡
  let <T, <T, t::f>> = x in
  if t = FileTypeRegular then lgc-lex-4 ( f , s ) else
  <writeln request ( 'Source file not found' )>]
```

```
[lgc-lex-4 ( f , s ) ≐
  let s = s[‘source’→f] in
  let e :: S = lgc-lex-source ( f , s )° in
  if e then lgc-error ( s , Sh , St ) else
  lgc-include-1 ( S )]
```

Extract the source from the input event  $x$ . Then prepare a state  $s$  for compilation.

### 4.3 Escape sequences

We now define quantities like `lgc-esc-#` for representing escape sequences. We also define a value, `lgc-EOF`, for representing the End Of File.

```
[lgc-EOF ≐ 0]
[lgc-esc-. ≐ ‘.’ – NULL]
[lgc-esc-- ≐ ‘-’ – NULL]
[lgc-esc-! ≐ ‘!’ – NULL]
[lgc-esc-# ≐ ‘#’ – NULL]
[lgc-esc-$ ≐ ‘$’ – NULL]
[lgc-esc-left ≐ ‘[’ – NULL]
[lgc-esc-right ≐ ‘]’ – NULL]
[lgc-esc-brace ≐ ‘}’ – NULL]
[lgc-esc-B ≐ ‘B’ – NULL]
[lgc-esc-C ≐ ‘C’ – NULL]
[lgc-esc-D ≐ ‘D’ – NULL]
[lgc-esc-N ≐ ‘N’ – NULL]
[lgc-esc-P ≐ ‘P’ – NULL]
[lgc-esc-R ≐ ‘R’ – NULL]
[lgc-esc-S ≐ ‘S’ – NULL]
[lgc-esc-f ≐ ‘f’ – NULL]
[lgc-esc-n ≐ ‘n’ – NULL]
[lgc-esc-r ≐ ‘r’ – NULL]
```

[lgc-esc-t  $\equiv$  't' - NULL]

[lgc-esc-x  $\equiv$  'x' - NULL]

## 4.4 Space trimming

The following functions remove spaces at the beginning or end of a string. They can remove at most one space character at each end. The input and output are lists of singleton strings.

[lgc-left-trim ( *a* )  $\equiv$   
    **if**  $a^h = \text{SP}$  **then**  $a^t$  **else**  $a$  ]  
    Remove leading space, if any.

[lgc-right-trim ( *a* )  $\equiv$   
    reverse ( lgc-left-trim ( reverse ( *a* ) ) ) ]  
    Remove trailing space, if any.

[lgc-trim ( *a* )  $\equiv$   
    lgc-right-trim ( lgc-left-trim ( *a* ) ) ]  
    Remove leading space, if any, and trailing space, if any.

## 4.5 Space contraction

The following functions contracts multiple spaces into single space characters and possibly trims or reverses the strings. The input and output are lists of singleton strings.

[lgc-reverse-contract ( *a* )  $\equiv$   
    lgc-reverse-contract1 ( *a* ,  $\top$  ) ]  
    Contract multiple spaces in *a* to single spaces and reverse *a*.

[lgc-reverse-contract1 ( *a* , *r* )  $\equiv$   
    **if**  $a \in \mathbf{A}$  **then** *r* **else**  
    **let**  $c :: a = a$  **in**  
    **if**  $c = \text{SP}$  **and**  $r^h = \text{SP}$  **then**  
    lgc-reverse-contract1 ( *a* , *r* ) **else**  
    lgc-reverse-contract1 ( *a* ,  $c :: r$  ) ]  
    Accumulate lgc-reverse-contract ( *a* ) in *r*.

[lgc-contract ( *a* )  $\equiv$   
    reverse ( lgc-reverse-contract ( *a* ) ) ]  
    Contract multiple spaces in *a* to single spaces.

```
[lgc-contract* ( a ) ≐
  if a ∈ A then T else
  lgc-contract ( ah ) :: lgc-contract* ( at )]
```

Apply lgc-contract ( e ) in each element of the list a.

```
[lgc-trim-contract ( a ) ≐
  lgc-left-trim ( reverse ( lgc-left-trim ( lgc-reverse-contract ( a ) ) ) )]
```

Contract multiple spaces in a to single spaces and remove leading and trailing spaces, if any.

## 4.6 Space trimming of text containing character positions

The following functions remove spaces at the beginning or end of a string. They can remove at most one space character at each end. The input and output are lists of singleton strings.

```
[lgc-lex-left-trim ( a ) ≐
  if ahh = SP then lgc-lex-left-trim ( at ) else a]
```

Remove leading spaces, if any.

```
[lgc-lex-right-trim ( a ) ≐
  reverse ( lgc-lex-left-trim ( reverse ( a ) ) )]
```

Remove trailing space, if any.

```
[lgc-lex-trim ( a ) ≐
  lgc-lex-right-trim ( lgc-lex-left-trim ( a ) )]
```

Remove leading space, if any, and trailing space, if any.

## 4.7 Space contraction of text containing character positions

The following functions contracts multiple spaces into single space characters and possibly trims or reverses the strings. The input and output are lists of singleton strings.

```
[lgc-lex-reverse-contract ( a ) ≐
  lgc-lex-reverse-contract1 ( a , T )]
```

Contract multiple spaces in a to single spaces and reverse a.

```
[lgc-lex-reverse-contract1 ( a , r ) ≐
  if a ∈ A then r else
  let c :: a = a in
  if ch = SP and rhh = SP then
  lgc-lex-reverse-contract1 ( a , r ) else
  lgc-lex-reverse-contract1 ( a , c :: r )]
```

Accumulate lgc-lex-reverse-contract ( a ) in r.

[lgc-lex-contract ( *a* )  $\equiv$   
reverse ( lgc-lex-reverse-contract ( *a* ) )]  
Contract multiple spaces in *a* to single spaces.

[lgc-lex-reverse-contract\* ( *a* )  $\equiv$   
**if** *a*  $\in$  **A** **then** **T** **else**  
lgc-lex-reverse-contract ( *a*<sup>h</sup> ) :: lgc-lex-reverse-contract\* ( *a*<sup>t</sup> )]  
Apply lgc-lex-contract ( *e* ) in each element of the list *a*.

[lgc-lex-contract\* ( *a* )  $\equiv$   
**if** *a*  $\in$  **A** **then** **T** **else**  
lgc-lex-contract ( *a*<sup>h</sup> ) :: lgc-lex-contract\* ( *a*<sup>t</sup> )]  
Apply lgc-lex-contract ( *e* ) in each element of the list *a*.

[lgc-lex-reverse-trim-contract ( *a* )  $\equiv$   
lgc-lex-left-trim ( lgc-lex-reverse-contract ( lgc-lex-left-trim ( *a* ) ) )]  
Contract multiple spaces in *a* to single spaces and remove leading and trailing spaces, if any. Reverse the list during the process.

[lgc-lex-trim-contract ( *a* )  $\equiv$   
reverse ( lgc-lex-reverse-trim-contract ( *a* ) )]  
Contract multiple spaces in *a* to single spaces and remove leading and trailing spaces, if any.

## 4.8 Lexical analysis

Split source into constituents and store them in *s*['page'], *s*['bib'], *s*['def'], *s*['body'].

[lgc-lex-source ( *f* , *s* )  $\equiv$   
**let** *f* =  
**let** *f* =  
**let** *f* =  
**let** *f* =  
**let** *f* =  
lgc-lex-position ( *f* , 0 ) **in**  
lgc-lex-newline ( *f* , **T** ) **in**  
lgc-lex-comment1 ( *f* ) **in**  
lgc-lex-escape1 ( *f* ) **in**  
lgc-lex-collect ( *f* ) **in**  
**let** *g* = lgc-lex-left-trim ( lgc-lex-extract-body ( *f* ) ) **in**  
**let** *s* = *s*['body'  $\rightarrow$  *g*] **in**  
**let** *s* = lgc-lex-extract-other ( *f* , *s* ) **in**  
*s*]

Apply lexical analysis to the source by adding positions, normalizing newline sequences, removing comments, replacing escape sequences

by escape codes, and collecting structures (like strings) into tokens. The funny way of stating let-statements ensures that the rather large intermediate results can be re-claimed by the garbage collector.

## 4.9 Add positions to text

```
[lgc-lex-position ( f , p ) ≡
  if f ∈ A then T else
  ⟨fh, p⟩ :: lgc-lex-position ( ft , p + 1 )]
Add position numbers to all characters.
```

## 4.10 Replace newline sequences with newline characters

Input files are assumed to be encoded in Logiweb Unicode UTF-8. This is identical to the Unicode UTF-8 encoding with the following, additional conventions concerning “special characters”, i.e. characters with codes between Code 0 and Code 31, inclusive:

**Code 9 horizontal tab (TAB)** A TAB is treated as a Code 32 space.

**Code 10 line feed (LF)** An LF marks the end of a line.

**Code 12 form feed (FF)** An FF is treated as an LF.

**Code 13 carriage return (CR)** A CR is treated as an LF. However, an LF which follows a non-ignored CR is ignored and a CR which follows a non-ignored LF is ignored. As an example, the sequence ⟨LF, CR, CR, LF⟩ is interpreted as a non-ignored LF, an ignored CR, a non-ignored CR, and an ignored LF. The non-ignored LF and CR both mark the end of a line. Hence, ⟨LF, CR, CR, LF⟩ is interpreted as two LF characters.

**Other** All characters in the range 0–31 other than TAB, LF, FF, and CR are considered illegal.

```
[lgc-lex-newline ( f , C ) ≡
  if f ∈ A then T else
  let ⟨c, p⟩ :: f = f in
  if c ≥ ‘ ’ then ⟨c, p⟩ :: lgc-lex-newline ( f , T ) else
  if c = C then lgc-lex-newline ( f , T ) else
  if c = LF then ⟨c, p⟩ :: lgc-lex-newline ( f , CR ) else
  if c = CR then ⟨LF, p⟩ :: lgc-lex-newline ( f , LF ) else
  if c = FF then ⟨LF, p⟩ :: lgc-lex-newline ( f , T ) else
  if c = TAB then ⟨SP, p⟩ :: lgc-lex-newline ( f , T ) else (p :: ‘Illegal
  character: 0x’ :: lgc-string2mixed ( c ))]
Normalize newline sequences in the text f into newline characters.
```

## 4.11 Remove comments

Logiweb sources may contain short and long comments.

A short comment starts with a multiquote followed by a semicolon and ends by a newline or the end of the file. Form feed (FF) and carriage return (CR) characters are converted to newlines, so a short comment may effectively be ended by an FF or CR.

A long comment starts with a multiquote followed by a left brace and ends by a multiquote followed by a right brace. The number of double quote characters of the closing escape sequence must match the number of double quote characters of the opening escape sequence.

All escape sequences inside short and long comments are ignored, except that a right brace escape sequence of the right size ends a long comment. In normal usage, the programmer is supposed to use escape sequences starting with two double quote characters. But a programmer may occasionally want to comment out a piece of source text which contains a mix of comments, code, and strings, in which case a long comment of starting with three double quote characters may be used.

```
[lgc-lex-comment1 ( f ) ::=
  if f ∈ A then T else
  if fhh ≠ QQ then fh :: lgc-lex-comment1 ( ft ) else
  lgc-lex-comment2 ( fh1 , 1 , ft )]
```

Remove comments from the text  $f$ .

```
[lgc-lex-comment2 ( P , n , f ) ::=
  if f ∈ A then repeat ( n , ⟨QQ, P⟩ ) else
  let ⟨c, p⟩ :: f = f in
  if c = QQ then lgc-lex-comment2 ( P , n + 1 , f ) else
  if n > 1 and c = ';' then lgc-lex-comment3 ( f ) else
  if n > 1 and c = '{' then lgc-lex-comment4 ( P , n , 0 , f ) else
  append ( repeat ( n , ⟨QQ, P⟩ ) , ⟨c, p⟩ :: lgc-lex-comment1 ( f ) )]
```

Parse the start of a comment. Invoke `lgc-lex-comment3 ( f )` for `""`; comments and `lgc-lex-comment4 ( P , n , 0 , f )` for `""{ comments.`

```
[lgc-lex-comment3 ( f ) ::=
  if f ∈ A then T else
  if fhh = LF then lgc-lex-comment1 ( ft ) else
  lgc-lex-comment3 ( ft )]
```

Remove all characters up to next newline.

```
[lgc-lex-comment4 ( P , n , m , f ) ::=
  if f ∈ A then ( P :: 'End of file in long comment' )• else
  if fhh = QQ then lgc-lex-comment4 ( P , n , m + 1 , ft ) else
  if m = n and fhh = '}' then lgc-lex-comment1 ( ft ) else
  lgc-lex-comment4 ( P , n , 0 , ft )]
```

## 4.12 Parse escape sequences

```
[lgc-lex-escape1 ( f ) ≡  
  if f ∈ A then T else  
  if fhh = QQ then lgc-lex-escape2 ( fh1 , ft ) else  
  fh :: lgc-lex-escape1 ( ft )]
```

Convert escape sequences in the text  $f$ .

```
[lgc-lex-escape2 ( P , f ) ≡  
  if f ∈ A then ⟨QQ, P⟩ :: T else  
  if fhh = QQ then lgc-lex-escape3 ( P , ft ) else  
  ⟨QQ, P⟩ :: lgc-lex-escape1 ( f )]
```

Convert the text  $f$  which follows one double quote.

```
[lgc-lex-escape3 ( P , f ) ≡  
  if f ∈ A then (P :: 'End of file in escape sequence')• else  
  let ⟨c⟩ :: f = f in  
  if c = QQ then lgc-lex-escape3 ( P , f ) else  
  ⟨c - NULL, P⟩ :: lgc-lex-escape1 ( f )]
```

Convert the text  $f$  which follows a multiquote.

## 4.13 Collect structures

```
[lgc-lex-collect ( f ) ≡  
  if f ∈ A then T else  
  let ⟨c, P⟩ :: f = f in  
  if c < NULL then lgc-lex-collect1 ( c , P , f ) else  
  if c = SP then ⟨SP, P⟩ :: lgc-lex-space ( f ) else  
  if c = LF then ⟨SP, P⟩ :: lgc-lex-space ( f ) else  
  if c = QQ then lgc-lex-string ( lgc-esc-- , P , T , f ) else  
  ⟨c, P⟩ :: lgc-lex-collect ( f )]
```

Collect tokens in the text  $f$ .

```
[lgc-lex-collect1 ( c , P , f ) ≡  
  if c = lgc-esc-- then lgc-lex-string ( lgc-esc-- , P , T , f ) else  
  if c = lgc-esc-# then lgc-lex-string ( lgc-esc-# , P , T , f ) else  
  if c = lgc-esc-$ then lgc-lex-string ( lgc-esc-$ , P , T , f ) else  
  if c = lgc-esc-P then lgc-lex-page ( P , T , T , f ) else  
  if c = lgc-esc-R then lgc-lex-ref ( P , T , T , f ) else  
  if c = lgc-esc-D then lgc-lex-def ( f ) else  
  if c = lgc-esc-B then lgc-lex-collect ( f ) else  
  if c = lgc-esc-. then (lgc-esc-- :: P :: ") :: lgc-lex-collect ( f ) else  
  if c = lgc-esc-S then ⟨lgc-esc-S, P⟩ :: lgc-lex-collect ( f ) else  
  if c = lgc-esc-N then ⟨lgc-esc-N, P⟩ :: lgc-lex-collect ( f ) else  
  if c = lgc-esc-C then ⟨lgc-esc-C, P⟩ :: lgc-lex-collect ( f ) else
```

( $P :: \text{'Unknown escape in body: 0x'} :: \text{lgc-string2mixed} ( c + \text{NULL} )$ )<sup>•</sup>]

Collect tokens that start with escape sequences in the text  $f$ .

#### 4.14 Space parsing

```
[lgc-lex-space ( f ) ≡
  let c = fhh in
  if c = SP or c = LF then lgc-lex-space ( ft ) else
  lgc-lex-collect ( f )]
Skip spaces.
```

#### 4.15 Page parsing

```
[lgc-lex-page ( P , r , R , f ) ≡
  if f ∈ A then lgc-lex-page1 ( P , r , R , T ) else
  let ⟨c, p⟩ :: f = f in
  if c = LF or c = lgc-esc-n then lgc-lex-page1 ( P , r , R , f ) else
  if c = QQ then lgc-lex-page ( P , T , r :: R , f ) else
  if c ≥ NULL then lgc-lex-page ( P , c :: r , R , f ) else
  (p :: 'Unknown escape in page: 0x' :: lgc-string2mixed ( c + NULL
  ))•]
```

Parse page directive until the end of the line. Chop at quotes. Accumulate characters in  $r$  and accumulate interquote strings in  $R$ . Finally call  $\text{lgc-lex-page1} ( P , r , R , f )$  where  $r$  is the page name and  $R$  is the list of prefixes.

```
[lgc-lex-page1 ( P , r , R , f ) ≡
  let r = lgc-lex-reverse-trim-contract ( r ) in
  let R = lgc-lex-reverse-contract* ( R ) in
  if r then ( P :: 'Empty page name'• ) else
  (lgc-esc-P :: P :: r :: R) :: lgc-lex-collect ( f )]
```

Normalize the page name  $r$  and the prefixes  $R$ . Then continue lexical analysis.

#### 4.16 Reference parsing

```
[lgc-lex-ref ( P , r , R , f ) ≡
  if f ∈ A then lgc-lex-ref1 ( P , r , R , T ) else
  let ⟨c, p⟩ :: f = f in
  if c = LF or c = lgc-esc-n then lgc-lex-ref1 ( P , r , R , f ) else
  if c = QQ then lgc-lex-ref ( P , T , r :: R , f ) else
  if c ≥ NULL then lgc-lex-ref ( P , c :: r , R , f ) else
  (p :: 'Unknown escape in reference: 0x' :: lgc-string2mixed ( c+NULL
  ))•]
```

Parse reference until the end of the line. Chop at quotes. Accumulate characters in  $r$  and accumulate interquote strings in  $R$ . Finally call  $\text{lgc-lex-page1} ( P , r , R , f )$  where  $r$  is the reference and  $R$  is the list of prefixes.

```
[lgc-lex-ref1 ( P , r , R , f ) ::=
  let r = lgc-lex-reverse-trim-contract ( r ) in
  let R = lgc-lex-reverse-contract* ( R ) in
  if r then ( P :: 'Empty reference' )• else
  (lgc-esc-R :: P :: r :: R) :: lgc-lex-collect ( f )]
```

Normalize the reference  $r$  and the prefixes  $R$ . Then continue lexical analysis.

## 4.17 Definition parsing

```
[lgc-lex-def ( f ) ::=
  if f ∈ A then T else lgc-lex-def0 ( fh1 , fh1 , T , T , f )]
```

```
[lgc-lex-def0 ( P , Q , r , R , f ) ::=
  if f ∈ A then lgc-lex-def2 ( P , Q , r , R , f ) else
  let ⟨c, p⟩ :: f = f in
  if c = LF or c = lgc-esc-n then
  lgc-lex-def0 ( P , fh1 , T , lgc-lex-def1 ( Q , r , R ) , f ) else
  if c ≥ NULL then lgc-lex-def0 ( P , Q , c :: r , R , f ) else
  if c = lgc-esc-P or c = lgc-esc-R or c = lgc-esc-D or c = lgc-esc-B then
```

```
lgc-lex-def2 ( P , Q , r , R , ⟨c, p⟩ :: f ) else ( p :: 'Unknown escape
in definition: 0x' :: lgc-string2mixed ( c + NULL ) )•]
```

Parse lines until the end of the definition section. Chop at newlines. Accumulate characters in  $r$  and accumulate lines in  $R$ . Finally call  $\text{lgc-lex-def2} ( P , Q , r , R , f )$  where  $r$  is the reference and  $R$  is the list of prefixes.  $P$  is the start of the definition directive and  $Q$  is the start of the line.

```
[lgc-lex-def1 ( Q , r , R ) ::=
  let r = reverse ( r ) in
  if R then ⟨r⟩ else
  let p = rh1 in
  let r = lgc-lex-trim-contract ( r ) in
  if r = T then R else
  if rt ≠ T or rhh ≠ QQ then r :: R else
  ( Q :: 'Construct must contain at least one proper character' )•]
```

```
[lgc-lex-def2 ( P , Q , r , R , f ) ::=
  let R = lgc-lex-def1 ( Q , r , R ) in
  let R = reverse ( R ) in
  (lgc-esc-D :: P :: R) :: lgc-lex-collect ( f )]
```

## 4.18 String parsing

```
[lgc-lex-string ( C , P , r , f ) ≡
  if f ∈ A then (P::‘End of file in string’)• else
  let ⟨c,p⟩::f = f in
  if c = QQ or c = lgc-esc-. then (C::P::vt2vector ( reverse ( r )
  ))::lgc-lex-collect ( f ) else
  if c ≥ NULL then lgc-lex-string ( C , P , c::r , f ) else
  if c = lgc-esc- then lgc-lex-string ( C , P , r , f ) else
  if c = lgc-esc-! then lgc-lex-string ( C , P , QQ::r , f ) else
  if c = lgc-esc-f then lgc-lex-string ( C , P , FF::r , f ) else
  if c = lgc-esc-n then lgc-lex-string ( C , P , LF::r , f ) else
  if c = lgc-esc-r then lgc-lex-string ( C , P , CR::r , f ) else
  if c = lgc-esc-t then lgc-lex-string ( C , P , TAB::r , f ) else
  if c = lgc-esc-x then lgc-lex-hex ( C , P , p , r , f ) else
  (p::‘Unknown escape in string: 0x’::lgc-string2mixed ( c + NULL
  ))•]
```

```
[lgc-lex-hex ( C , P , Q , r , f ) ≡
  if f ∈ A then (P::‘End of file in string’)• else let ⟨c,p⟩::f = f in
  if c = QQ then (Q::‘End of string in hex code’)• else
  if c = ‘.’ then lgc-lex-string ( C , P , r , f ) else
  if c = SP or c = LF then lgc-lex-hex ( C , P , Q , r , f ) else
  if ‘0’ ≤ c and c ≤ ‘9’ then lgc-lex-hex1 ( C , P , Q , c - ‘0’ , r ,
  f ) else
  if ‘A’ ≤ c and c ≤ ‘F’ then lgc-lex-hex1 ( C , P , Q , c - ‘A’ + Base
  , r , f ) else
  if c ≥ NULL then (p::‘Invalid character in hex constant’)• else
  (P::‘Unknown escape in string: 0x’::lgc-string2mixed ( c + NULL
  ))•]
```

```
[lgc-lex-hex1 ( C , P , Q , d , r , f ) ≡
  if f ∈ A then (P::‘End of file in string’)• else
  let ⟨c,p⟩::f = f in
  if c = QQ then (Q::‘End of string in hex code’)• else
  if c = ‘.’ then (Q::‘Odd number of digits in hex code’)• else
  if c = SP or c = LF then lgc-lex-hex1 ( C , P , Q , d , r , f
  ) else
  if ‘0’ ≤ c and c ≤ ‘9’ then lgc-lex-hex ( C , P , Q , NULL + d ·
  16 + c - ‘0’::r , f ) else
  if ‘A’ ≤ c and c ≤ ‘F’ then lgc-lex-hex ( C , P , Q , NULL + d ·
  16 + c - ‘A’ + Base::r , f ) else
  if c ≥ NULL then (p::‘Invalid character in hex constant’)• else
  (P::‘Unknown escape in string: 0x’::lgc-string2mixed ( c + NULL
  ))•]
```

## 4.19 Extract result of lexical analysis

```
[lgc-lex-extract-body ( f ) ≐
  if f ∈ A then T else
  let c = fhh in
  if c = lgc-esc-P or c = lgc-esc-R or c = lgc-esc-D then lgc-lex-extract-
  ( ft ) else
  if c = SP then lgc-lex-extract-body1 ( fh , ft ) else
  fh :: lgc-lex-extract-body ( ft )]
```

Extract the body from the text  $f$  by disregarding page, reference, and definition directives.

```
[lgc-lex-extract-body1 ( x , f ) ≐
  if f ∈ A then T else
  let c = fhh in
  if c = lgc-esc-P or c = lgc-esc-R or c = lgc-esc-D then lgc-lex-extract-
  ( x , ft ) else
  if c = SP then x :: lgc-lex-extract-body ( ft ) else
  x :: lgc-lex-extract-body ( f )]
```

Extract the body from the text  $f$  knowing that it follows the space character  $x$ .

```
[lgc-lex-extract-other ( f , s ) ≐
  if f ∈ A then s else
  let c = fhh in
  if c = lgc-esc-P then lgc-lex-extract-page ( f , s ) else
  if c = lgc-esc-R then lgc-lex-extract-ref ( f , s ) else
  if c = lgc-esc-D then lgc-lex-extract-def ( f , s ) else
  if c = lgc-esc-$ then lgc-lex-extract-include ( f , s ) else
  if c = lgc-esc-# then lgc-lex-extract-include ( f , s ) else
  lgc-lex-extract-other ( ft , s )]
```

```
[lgc-lex-extract-page ( f , s ) ≐
  if s[‘page’] ≠ T then (fh1 :: ‘More than one page name found’)• else
  let s = s[‘page’ → fhtt] in
  lgc-lex-extract-other ( ft , s )]
```

```
[lgc-lex-extract-ref ( f , s ) ≐
  let s = push ( s , ‘bib’ , fhtt ) in
  lgc-lex-extract-other ( ft , s )]
```

```
[lgc-lex-extract-def ( f , s ) ≐
  let ( T :: p :: r :: R ) :: f = f in
  let s = push ( s , ‘def’ , ( p :: r ) :: R ) in
  lgc-lex-extract-other ( f , s )]
```

```
[lgc-lex-extract-include ( f , s ) ≡
  let (T :: p :: n) :: f = f in
  let s = push ( s , 'nincludes' , p :: n ) in
  lgc-lex-extract-other ( f , s )]
```

## 4.20 Include processing

The `lgc-include-1 ( s )` function requests the include files and `lgc-include-2 ( x , s )` processes them.

```
[lgc-include-1 ( s ) ≡
  let l = s['nincludes'] in
  if l ∈ A then lgc-load-1 ( s ) else
  let (T :: n) :: l = l in
  if not s['includes'][n] then lgc-include-1 ( s['nincludes'→l] ) else
  let n = lgc-tilde-expand ( n , s['parameters'] ) in
  let s = lgc-progress ( 'Reading file:' :: n , 3 , s ) in
  let s = lgc-push-event ( s , fileTypeRead ( n ) ) in
  lgc-exec-events ( s , lgc-include-2 ( x , s ) )]
```

```
[lgc-include-2 ( x , s ) ≡
  let (T, <T, t :: f>) = x in
  let l = s['nincludes'] in
  let (p :: n) :: l = l in
  if t = FileTypeNonexistent then
  lgc-error ( s , p , 'Could not find include file ' :: n ) else
  let s = s['nincludes'→l] in
  let s = s['includes', n]⇒f in
  lgc-include-1 ( s )]
```

## 5 Loading

### 5.1 Load processing

The `lgc-load-1 ( s )` function requests all Logiweb pages transitively referenced by the page being translated.

The source text references other pages using `""R` escape sequences. The reference following the `""R` escape sequence may be:

A **url** such as `http://logiweb.eu/logiweb/page/base/fixed/vector/vector.lgw` or `file:~/logiweb/latest/base/rack.lgr`

A **page name** such as `name:base`

References which do not contain a colon character are taken to be page names, so `name:base` can be abbreviated as `base`. Names are looked up using the `namepath` as described later.

A url must have type `http:` or `file:` or `lgw:.` An http url is retrieved using an http GET operation. A file url is looked up in the local file system. Tilde expansion is performed on file urls. An lgw url is looked up using the `path` as described later.

An http or file url must have extension `.lgw` or `.lgr` or `.lgu`. The extension determines the format of the referenced page. A url with extension `.lgw`, `.lgr`, and `.lgu` contains a Logiweb vector, rack, and url, respectively. A Logiweb vector is a compact format suited for transmitting over the Internet which takes some time to translate to internal form. A Logiweb rack is a more extensive format suited for storing in the local file system which is faster to translate to internal format. A Logiweb url is like a symbolic link in that it just contains the http url of the page to be fetched.

As mentioned, names are translated to pages using the `namepath`. This is done by considering the elements of `namepath` one at a time and replacing the rightmost colon character of the element by the given name. The result must be a url which is then looked up. If successful, the page found is used. Otherwise, the next element of the `namepath` is tried.

Similarly, urls of type `lgw:` are looked up using the `path` variable by replacing the rightmost colon character of each element by the characters following `lgw:.` In this case, however, the characters following `lgw:` must be a Logiweb reference expressed in mixed endian hexadecimal. In mixed endian hexadecimal, bytes are written in network byte order and each byte is written as two hexadecimal digits with the most significant digit first.

Once a page is retrieved, it will be in either `.lgw` or `.lgr` format. If it is in `.lgw` format, then its reference is extracted and an attempt is made to look it up in `.lgr` format using the `path`. If that is successful, loading is based on the `.lgr` version.

Note that one should only retrieve pages in `.lgr` format from trusted locations and only over high bandwidth channels. In practice, most users will be happy just retrieving `.lgr` pages from their own cache. Whenever the Logiweb compiler has translated an `.lgw` page, it stores the `.lgr` equivalent in the users cache so that the time consuming translation is done once only.

Loading a page involves the following major steps:

**Fetch** Translate the reference to a page in `.lgw` or `.lgr` format.

**Trisect** Do a first round of processing of the page.

**Traverse** Transitively load all pages referenced by the page.

**Codify** Do a second round of processing of the page.

## 5.2 State entries

A number of hooks of the state are used for keeping track of the loading process:

`s['cluster'][r]` The cache of the page with reference *r*. This is the main result of loading.

- s[‘stack’] List of list of not yet loaded pages. The element at the bottom of the stack is a list of urls/names. The other elements are lists of references. Set to **T** when all pages are loaded.
- s[‘events’] List of output events in reverse order.
- s[‘path’] Search path (from path or namepath option).
- s[‘fetching’] The url being fetched.
- s[‘suffix’] The suffix of the url being fetched (lgw or lgr or lgu)
- s[‘type’] The type of the url being fetched (http or file)
- s[‘vector’] The vector of the page that has been fetched. This is set only when a page has been found in .lgw format but a search for the same page in .lgr format is in progress.
- s[‘refbib’] The bibliography as a list of references.

### 5.3 Loading in more detail

Loading of a Logiweb page is a process which transforms a Logiweb page into an in-memory *rack* structure suited for computer processing. A *rack* is an inhomogeneous array, i.e. an array  $c$  for which  $c[i]$  contains different kinds of information for different values of  $i$ . Values of indices  $i$  are named *hooks*, and we shall refer to  $c[i]$  as the value that hangs on the the hook  $i$ . Racks correspond to record structures in e.g. C. The input to the load process may be:

- The Logiweb reference of the page. The reference of a page is a sequence of about 30 bytes which provides a world-wide unique identification of the page.
- The page itself in lgw or lgr format.
- The name of a file which contain the page in lgw or lgr format.
- An http url which points to a file which contains the page in lgw format.
- The name of the page which, combinted with the namepath argument of the compiler allows to locate a file which contains the page in lgw or lgr format.

When given a Logiweb reference as input, loading comprises the following steps:

**Fetch** Translate the reference of a page into the vector of the page. The vector of the page is a sequence of bytes which encodes the page in a compact way suited for transmission over a network. Fetching a page comprises two activities:

**Locate** Translate the reference of a page into an http URL suited for getting the page.

**Get** Receive the vector using an http GET operation.

**Trisect** Parse the vector of the page into a bibliography, a dictionary, and a flat body. The bibliography is a sequence of references. Reference zero (the first reference) of the bibliography is the reference of the page itself. The proper references (all but the first one) are pointers to other Logiweb pages. The dictionary is an association list which maps indexes to arities. Indexes as well as arities are natural numbers. A flat body is a sequence of unparsed bytes.

**Traverse** Recursively load the proper references of the bibliography.

**Codify** Parse the flat body, macro expand the body and harvest *revelations* (i.e. *definitions*, *introductions*, and *proclamations*) from the expanded tree, then verify the correctness of the page. We shall refer to the outcome of macro expansion and harvesting as the *expansion* and *codex*, respectively, of the page. The expansion resembles an S-expression like the body does. The codex is an associative structure containing all revelations in the expansion organized in a way that makes the revelations easy to access. We shall refer to the outcome of verification as a *diagnose* which, if empty, indicates that the page is correct. Codification comprises the following activities:

**Unpack** Parse the flat body and return a parse tree. We shall refer to the parse tree as the *body* of the page. The body resembles a Lisp S-expression.

**Initialize** Construct an initial codex. The initial codex is empty if the bibliography contains more than one reference (i.e. contains at least one proper reference). Otherwise, the codex contains exactly one entry which proclaims a *proclamation* symbol. The user can use the proclamation symbol to assign semantics to other symbols and thus bootstrap the Logiweb system. This is for experienced, determined users only. Other users just reference at least one other page and thus leaves the bootstrap to somebody else.

**Compile** Compile all value definitions in the codex and lazily verify the page. During the first iteration of the codification there are no value definitions in the codex so the compilation done in the first iteration is always trivial.

**Macro expand** Macro expand the page. First look for a macro expander in the codex of the page. If one is found, apply it. Otherwise, look for a macro expander in the first reference of the page. If one is found, apply it. Otherwise, use the identity function as macro expander, i.e. use the body unchanged as expansion.

**Harvest** Scan the expansion for revelation symbols, i.e. for symbols which are proclaimed to denote definition, introduction, or proclamation. Each time a revelation symbol is found, add one revelation to the codex. When encountering a symbol which is proclaimed to be a hiding symbol, do not scan the subtrees of the symbol. If harvesting changes the codex, reiterate from *Compile* above.

After loading a page, one may do the following:

- Render the page. Rendering may result in a human readable version of the page as well as executables.
- Verify the page. Verification is done lazily during codification, so one just needs to force evaluation of the diagnose to do verification.
- Dump the page to cache. Before a page can be cached, verification has to be forced.

## 5.4 Top level function

The `lgc-load-1 ( s )` function implements traversing in that it arranges that all transitively referenced pages are loaded. When invoked, lexical analysis has placed a structure of form  $\langle n :: p, \dots \rangle$  in `s['bib']` where  $n$  is the name of a directly referenced page and  $p$  is a list of prefixes for grammar construction. The `s['bib']` list is in reverse order compared to the source file, so first task is to reverse it. Then `lgc-heads ( b )` is used to extract the names, and the list of names is stacked. During loading, this list of names appear at the bottom of the stack. All other elements of the stack are lists of references. The actual stacking of `lgc-heads ( b )` is done by `lgc-load-fetch0 ( (lgc-heads ( b ) ) , s )`.

The `lgc-load-setpath ( s )` function sets `s['path']` to the path relevant for the first element of the top element of the stack. When there is more than one element in the stack, the top element is a list of references and, hence, the relevant path is `s['parameters']['path']`. If the stack has one element then, usually, the relevant path is `s['parameters']['namepath']`. If the first element of the top element is an explicit Logiweb reference (starting with 'lgw:') then the relevant path is `s['parameters']['path']`. The `lgc-load-setpath ( s )` function is defined separately because it is needed more than one place.

```
[lgc-load-1 ( s ) ≡
  let b = reverse ( s['bib'] ) in
  let s = s['bib'→b] in
  lgc-load-fetch0 ( (lgc-heads ( b ) ) , s )]
```

Reverse the bibliography  $b$  from lexical analysis, extract page names, and call `lgc-load-fetch0 ( S , s )` to stack the list of page names and invoke fetching.

[lgc-load-fetch0 (  $S$  ,  $s$  )  $\stackrel{\bullet\bullet}{=}$

**let**  $s = s[\text{'stack'} \rightarrow S]$  **in**  
**let**  $s = \text{lgc-load-setpath} ( s )$  **in**  
 $\text{lgc-load-fetch} ( s )]$

Initialize the stack and leave further work to  $\text{lgc-load-fetch} ( s )$ .

[lgc-load-setpath (  $s$  )  $\stackrel{\bullet\bullet}{=}$

**let**  $r = s[\text{'stack'}]$  **in**  
**let**  $p = r^{\text{hh}}$  **in**  
**if**  $p$  **then**  $s$  **else**  
**let**  $s = \text{lgc-progress-fetch} ( p , r , s )$  **in**  
**if**  $r^{\text{t}} \in \mathbf{P}$  **or**  $\text{lgc-prefix} ( \text{lgc-lgw-prefix} , p )$  **then**  
 $s[\text{'path'} \rightarrow s[\text{'parameters'}][\text{'path'}]]$  **else**  
 $s[\text{'path'} \rightarrow s[\text{'parameters'}][\text{'namepath'}]]]$

Set  $s[\text{'path'}]$  to path or namepath argument as appropriate.

[lgc-progress-fetch (  $p$  ,  $r$  ,  $s$  )  $\stackrel{\bullet\bullet}{=}$

**if**  $r^{\text{t}}$  **then**  $\text{lgc-progress} ( \text{'Fetching ' } :: p , 3 , s )$  **else**  
**let**  $P = \text{'lgw:' } :: \text{lgc-string2mixed} ( p )$  **in**  
**if**  $s[\text{'cluster'}][p] = \mathbf{T}$  **then**  
 $\text{lgc-progress} ( \text{'Fetching ' } :: P , 3 , s )$  **else**  
 $\text{lgc-progress} ( \text{'Fetching ' } :: P , 4 , s )]$

[lgc-heads (  $b$  )  $\stackrel{\bullet\bullet}{=}$

**if**  $b \in \mathbf{A}$  **then**  $\mathbf{T}$  **else**  $b^{\text{hh}} :: \text{lgc-heads} ( b^{\text{t}} )]$

Extract the references from the bibliography  $b$ .

## 5.5 Constants

[lgc-file-prefix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'file:'} )]$

[lgc-http-prefix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'http:'} )]$

[lgc-//-prefix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'//'} )]$

[lgc-lgw-prefix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'lgw:'} )]$

[lgc-name-prefix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'name:'} )]$

[lgw-suffix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'lgw'} )]$

[lgr-suffix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'lgr'} )]$

[lgu-suffix  $\stackrel{\bullet\bullet}{=}$   $\text{vt2vector*} ( \text{'lgu'} )]$

## 5.6 Auxiliary functions

```
[lgc-tilde-expand1 ( n , s ) ≐  
  if n then T else  
  if nh ≠ '~' then n else  
  s['parameters'][home]::nt]
```

Tilde expansion when  $n$  is a list of singleton strings.

```
[lgc-cwd-expand ( n , s ) ≐  
  let n = vt2vector* ( n ) in  
  if nh = '/' then n else s['cwd']::'/'::n]
```

Expand relative pathname into absolute pathname. If combined with tilde expansion, do tilde expansion first.

```
[lgc-prefix ( x , y ) ≐  
  if x ∈ A then T else  
  if y ∈ A then F else  
  xh = yh and lgc-prefix ( xt , yt )]
```

Return T if the list  $x$  is a prefix of the list  $y$ .

```
[lgc-char-split ( c , a ) ≐ lgc-char-split1 ( c , a , T )]
```

Split the list  $a$  at the element with value  $c$  and return a pair  $u::v$  of the list preceding and succeeding  $c$ . If  $c$  does not occur in  $a$  then  $u$  is set to  $a$  and  $v$  to T.

```
[lgc-char-split1 ( c , a , r ) ≐  
  if a ∈ A then reverse ( r )::T else  
  let C::a = a in  
  if c = C then reverse ( r )::a else  
  lgc-char-split1 ( c , a , C::r )]
```

Compute  $u::v = \text{lgc-char-split} ( c , a )$ , accumulating  $u$  in  $r$  in reverse order.

```
[lgc-replace-colon ( n , r ) ≐  
  vt2vector* ( lgc-replace-colon1 ( reverse ( n ) , r ) )]
```

Replace the rightmost colon character in  $n$  by  $r$ .

```
[lgc-replace-colon1 ( n , r ) ≐  
  if n then • else  
  let c::n = n in  
  if c = ':' then reverse ( n )::r else  
  lgc-replace-colon1 ( n , r )::c]
```

Auxiliary function for computing  $\text{lgc-replace-colon} ( n , r )$ .

```
[lgc-file-suffix ( n ) ≐  
  let n = reverse ( n ) in
```

```
let u :: v = lgc-char-split ( '.', n ) in
reverse ( u )]
```

## 5.7 Message generators

```
[lgc-load-fetch-ref-failed ( r , s ) ≡
```

```
lgc-simple-error ( 'Could not load reference ' :: r , s )]
```

Complain about a reference which could not be loaded. *r* is supposed to be the name of the reference as given in the source file or the mixed endian hexadecimal representation of an indirectly referenced page.

```
[lgc-load-no-colon ( s ) ≡
```

```
let n = s[‘path’]h in
```

```
lgc-simple-error ( 'Missing colon in path or namepath element ' :: n , s )]
```

All elements of the path and namepath parameters must contain at least one colon character.

```
[lgc-wrong-suffix ( n , s ) ≡
```

```
let s = lgc-progress ( 'Illegal suffix in path or namepath element ' :: n , 1 , s ) in
```

```
lgc-simple-error ( 'Only .lgw, .lgr, and .lgu are legal suffixes' , s )]
```

```
[lgc-wrong-lgu-suffix ( n , s ) ≡
```

```
let s = lgc-progress ( 'Illegal suffix in lgu link: ' :: n , 1 , s ) in
```

```
let s = lgc-progress ( 'Only .lgw and .lgu are legal suffixes' , 1 , s ) in
```

```
lgc-simple-error ( 'Source of link: ' :: s[‘type’] :: ‘:’ :: s[‘fetching’] , s )]
```

All elements of the path and namepath parameters must contain at least one colon character.

```
[lgc-load-malformed-url ( n , s ) ≡
```

```
lgc-simple-error ( 'Malformed url: http:' :: n , s )]
```

Complain about a malformed http url. Some valid http urls read:

- //my.domain:8080/my/path.lgw
- //my.domain/my/path.lgr

```
[lgc-load-malformed-lgu ( n , s ) ≡
```

```
let s = lgc-progress ( 'Malformed lgu link: ' :: n , 1 , s ) in
```

```
let s = lgc-progress ( 'Such links must start with 'http:' , 1 , s ) in
```

```
lgc-simple-error ( 'Source of link: ' :: s[‘type’] :: ‘:’ :: s[‘fetching’] , s )]
```

Complain about a malformed lgu link.

```
[lgc-load-malformed-page ( s ) ≡
  lgc-simple-error ( 'Malformed page found at ' :: s['type'] :: ':' :: s['fetching']
    , s )]
  Complain about a malformed page.
```

```
[lgc-load-wrong-page ( s ) ≡
  lgc-simple-error ( 'Wrong page found at ' :: s['type'] :: ':' :: s['fetching']
    , s )]
  Complain about a malformed page.
```

## 5.8 Requesting pages

The `lgc-load-fetch ( s )` function initiates loading of the first element of the top element of the stack (if any). When done, the function invokes `lgc-grammar ( s )`.

When the stack has more than one element, the top element of the stack is a list of references. In that case, the first element of the top element is loaded using a call to `lgc-load-fetch-ref ( r , s )`. Otherwise, `lgc-load-fetch-name ( s )` is used. That function dispatches on the url type of the reference which may be 'file:', 'http:', 'lgw:', or 'name:', defaulting to 'name:'.

If the reference is given as an url of type 'file:' or `http :` then `lgc-load-fetch-file ( r , s )` or `lgc-load-fetch-http ( r , s )` is called. Otherwise, the reference is looked up using the path. If the path is empty, it has been exhausted. Otherwise, the next element of the path is tried. When the next element of the path is tried, the rightmost colon character of the path element is replaced by the name or reference to be looked up, and `lgc-load-fetch-path ( n , s )` is called which dispatches on the url type of the resulting url.

```
[lgc-load-fetch ( s ) ≡
  let r = s['stack'] in
  if rh and rt then lgc-grammar ( s['stack'→T] ) else
  let v = s['vector'] in
  if not v then lgc-load-fetch-ref ( lgw-parse-string ( v )h , s ) else
  if rh then lgc-load-codify ( s ) else
  if rt then lgc-load-fetch-name ( s ) else
  if not s['cluster'][rhh] then lgc-load-fetch0 ( rht :: rt , s ) else
  lgc-load-fetch-ref ( rhh , s )]
```

Find out whether or not we are done with loading. If we are done, invoke `lgc-grammar ( s )`. Otherwise, if we have already found the page in .lgw format, look for the page in .lgr format. Otherwise, if we are at the bottom of the stack, fetch a url/name, else fetch a reference.

```
[lgc-load-fetch-name ( s ) ≡
  let n = s['stack']hh in
  if lgc-prefix ( lgc-file-prefix , n ) then
```

```

lgc-load-fetch-file ( list-suffix ( n , 5 ) , s ) else
if lgc-prefix ( lgc-http-prefix , n ) then
lgc-load-fetch-http ( list-suffix ( n , 5 ) , s ) else
if lgc-prefix ( lgc-lgw-prefix , n ) then
lgc-load-fetch-ref1 ( list-suffix ( n , 4 ) , s ) else
if lgc-prefix ( lgc-name-prefix , n ) then
lgc-load-fetch-ref1 ( list-suffix ( n , 5 ) , s ) else
lgc-load-fetch-ref1 ( n , s )

```

Dispatch on url type.

```

[lgc-load-fetch-ref ( r , s ) ≡
lgc-load-fetch-ref1 ( lgc-string2mixed ( r ) , s )]

```

Fetch an indirectly referenced page. Indirectly referenced pages are always looked up using the mixed endian hexadecimal representation of the Logiweb reference of the page.

```

[lgc-load-fetch-ref1 ( r , s ) ≡
let p = s['path'] in
let v = s['vector'] in
if p and v then lgc-load-fetch-ref-failed ( r , s ) else
if p then lgc-load-receive-lgw1 ( v , s['vector'→T] ) else
let n'::p = p in
let e::n = lgc-replace-colon ( vt2vector* ( n' ) , r )o in
if e then lgc-load-no-colon ( s ) else
let t = lgc-file-suffix ( n ) in
if t ≠ lgw-suffix and t ≠ lgr-suffix and t ≠ lgu-suffix then
lgc-wrong-suffix ( n' , s ) else
let s = s['suffix'→t] in
let s = s['path'→p] in
if not v and t ≠ lgr-suffix then lgc-load-fetch-ref1 ( r , s ) else
lgc-load-fetch-path ( n , s )]

```

Locate page using  $s[\text{'path'}]$  where  $s[\text{'path'}]$  is the namepath parameter if we are at the bottom of the stack and the path parameter otherwise. In the latter case,  $r$  is always a mixed endian hexadecimal logiweb reference. The function tries the next element of the path, if any. When the path is exhausted, the function checks if we have the page in .lgw format. If we have, then the function translates the .lgw format. If not, then the function issues an error message. When the path is not exhausted, the function tries the next path element but only accepts elements with suffix .lgr if we already have the page in .lgw format.

```

[lgc-load-fetch-path ( n , s ) ≡
if lgc-prefix ( lgc-file-prefix , n ) then
lgc-load-fetch-file ( list-suffix ( n , 5 ) , s ) else
if lgc-prefix ( lgc-http-prefix , n ) then

```

```
lgc-load-fetch-http ( list-suffix ( n , 5 ) , s ) else
lgc-load-fetch-file ( n , s )]
```

After replacing the rightmost colon of the next path element by the name being looked up, dispatch on url type (`file:` of `http:`, defaulting to `file:`).

```
[lgc-load-fetch-file ( n , s ) **
let s = s[‘fetching’→n] in
let s = s[‘type’→file] in
let n = lgc-tilde-expand1 ( n , s ) in
let s = lgc-progress ( ‘Reading file:’:n , 4 , s ) in
let s = lgc-push-event ( s , fileTypeRead ( n ) ) in
lgc-exec-events ( s , lgc-load-receive ( x , s ) )]
```

Request given file.

```
[lgc-load-fetch-http ( n , s ) **
let s = s[‘fetching’→n] in
let s = s[‘type’→http] in
let s = lgc-progress ( ‘Reading http:’:n , 4 , s ) in
if not lgc-prefix ( lgc-//-prefix , n ) then
lgc-load-malformed-url ( n , s ) else
let d :: q = lgc-char-split ( ‘/’ , list-suffix ( n , 2 ) ) in
let d :: p = lgc-char-split ( ‘:’ , d ) in
let p = if p then ‘80’ else p in
let x :: p = lgc-atoi ( p , T )o in
if x or p < 0 or p > 65535 then
lgc-load-malformed-url ( n , s ) else
let s = lgc-push-event ( s , tcpQuery ( d , p , 3 , 0 , GET/::q )
) in
lgc-exec-events ( s , lgc-load-receive ( x , s ) )]
```

Request given http url with a patience of  $3 \cdot 10^0$  seconds.

## 5.9 Receiving pages

The `lgc-load-receive ( x , s )` may or may not receive a page. If it receives nothing, it calls `lgc-load-received-nothing ( s )`. Otherwise, it dispatched on the type of the received data.

```
[lgc-load-receive ( x , s ) ** let ⟨T,⟨T,x⟩⟩ = x in
let t = s[‘type’] in
if t = ‘http’ and x or t = ‘file’ and xh = FileTypeNonexistent then
lgc-load-received-nothing ( s ) else
let x = if t = ‘file’ then xt else x in
let t = s[‘suffix’] in
if t = lgw-suffix then lgc-load-receive-lgw ( x , s ) else
if t = lgr-suffix then lgc-load-receive-lgr ( x , s ) else
```

**if**  $t = \text{lgu-suffix}$  **then**  $\text{lgc-load-receive-lgu} ( x , s )$  **else**  
 $\text{lgc-panic} ( \text{'Internal error in lgc-load-receive'} )$ ]

[ $\text{lgc-load-received-nothing} ( s ) \equiv$   
**let**  $n = s[\text{'stack'}]^{\text{hh}}$  **in**  
**if**  $\text{lgc-prefix} ( \text{lgc-file-prefix} , n )$  **or**  $\text{lgc-prefix} ( \text{lgc-http-prefix} , n )$   
**then**  
 $\text{lgc-load-fetch-ref-failed} ( n , s )$  **else**  
 $\text{lgc-load-fetch} ( s )$ ]

[ $\text{lgc-load-receive-lgw} ( x , s ) \equiv$   
**let**  $e :: r :: T = \text{lgw-parse-string} ( x )^\circ$  **in**  
**if**  $e$  **or**  $r = \text{'}$  **or**  $\text{vector-index} ( r , 0 ) \neq \text{lgc-logiweb-version}$  **then**  
 $\text{lgc-load-received-nothing} ( s )$  **else**  
**let**  $( R :: B ) :: S = s[\text{'stack'}]$  **in**  
**if**  $\text{lgc-wrong-ref} ( r , R , S )$  **then**  $\text{lgc-load-wrong-page} ( s )$  **else**  
**if**  $S \neq T$  **or**  $\text{lgc-prefix} ( \text{lgc-lgw-prefix} , R )$  **then**  $\text{lgc-load-receive-lgw1}$   
 $( x , s )$  **else**  
**if**  $s[\text{'cluster'}][r] \neq T$  **then**  $\text{lgc-load-fetch0} ( B :: S , \text{push} ( s , \text{'refbib'}$   
 $, r ) )$  **else**  
**let**  $s = s[\text{'vector'} \rightarrow x]$  **in**  
**let**  $s = s[\text{'path'} \rightarrow s[\text{'parameters'}][\text{'path'}]]$  **in**  
 $\text{lgc-load-fetch} ( s )$ ]

Receive the fetched page in .lgw format. Parse the reference of the received page. If the version number is wrong, consider the page non-existent. Otherwise check that the page is the one requested and complain if not. If the page is requested by reference, go on load it. Otherwise, it is requested by name in which case it might already be in the cluster or may be available in lgr format. First check if it is in the cluster and go on if it is. Otherwise, search for the page in lgr format.

[ $\text{lgc-load-receive-lgw1} ( x , s ) \equiv$   
**let**  $e :: c = \text{lgw-trisect} ( x )^\circ$  **in**  
**if**  $e$  **then**  $\text{lgc-load-malformed-page} ( s )$  **else**  
**let**  $r = c[0]$  **in**  
**let**  $T :: b = c[r][\text{'bibliography'}]$  **in**  
**let**  $( R :: B ) :: S = s[\text{'stack'}]$  **in**  
**let**  $s = \text{if } S \text{ then } \text{push} ( s , \text{'refbib'} , r ) \text{ else } s$  **in**  
**let**  $s = s[(\text{'cluster'}, r) \Rightarrow c]$  **in**  
**let**  $n = \text{if } S \text{ then } R \text{ else 'lgw:' :: lgc-string2mixed} ( r )$  **in**  
 $\text{lgc-load-fetch0} ( b :: ((r :: \text{'lgw'} :: n) :: B) :: S , s )$ ]

Receive a page in lgw format, knowing that the reference is the one requested and that there is no easy way to process the page. In this case, trisect the page and fetch the pages in the bibliography.

[lgc-logiweb-version  $\equiv$  1]

The current version of Logiweb.

[lgc-wrong-ref (  $r$  ,  $R$  ,  $S$  )  $\equiv$

**if not**  $S$  **then**  $r \neq R$  **else**

**if not** lgc-prefix ( lgc-lgw-prefix ,  $R$  ) **then** F **else**

list-suffix (  $R$  , 4 )  $\neq$  lgc-string2mixed (  $r$  )]

The lgc-wrong-ref (  $r$  ,  $R$  ,  $S$  ) function returns T if the references  $r$  and  $R$  differ. If we are at the top of the stack (i.e. if  $S$  is empty) then  $R$  is a bibliographic reference which can only be compared to  $r$  if it is of type 'lgw:'.

[lgc-load-receive-lgr (  $x$  ,  $s$  )  $\equiv$

**let**  $s = s$ [vector'  $\rightarrow$  T] **in**

**let**  $e :: c = \text{sl2rack} ( x )^\circ$  **in**

**if**  $e$  **then** lgc-load-malformed-page (  $s$  ) **else**

**let**  $r :: b = c$ [bibliography'] **in**

**let**  $c = T$ [0  $\rightarrow r$ ][ $r \rightarrow c$ ] **in**

**let** (  $R :: B$  ) ::  $S = s$ [stack'] **in**

**if** lgc-wrong-ref (  $r$  ,  $R$  ,  $S$  ) **then** lgc-load-wrong-page (  $s$  ) **else**

**let**  $s = \text{if } S \text{ then push} ( s , \text{'refbib'} , r ) \text{ else } s$  **in**

**if not**  $s$ [cluster'] [ $r$ ] **then** lgc-load-fetch0 (  $B :: S$  ,  $s$  ) **else**

**let**  $s = s$ [('cluster',  $r$ )  $\Rightarrow c$ ] **in**

**let**  $n = \text{if } S \text{ then } R \text{ else 'lgw:'} :: \text{lgc-string2mixed} ( r )$  **in**

lgc-load-fetch0 (  $b :: ((r :: \text{'lgw}' :: n) :: B) :: S$  ,  $s$  )]

Receive the fetched page in .lgr format. Clear  $s$ [vector'] to indicate that we no longer try to find the page in .lgr format. Unpack the rack of the fetched page, convert it to a cache, and store it in the cluster. Then unstack the page that has been found, stack the type of the page (.lgw) and stack the bibliography. Then go on fetching pages.

[lgc-load-receive-lgu (  $n$  ,  $s$  )  $\equiv$

**let**  $n = \text{reverse} ( \text{lgc-trim-newline} ( \text{reverse} ( n ) ) )$  **in**

**let**  $t = \text{lgc-file-suffix} ( n )$  **in**

**if**  $t \neq \text{lgw-suffix}$  **and**  $t \neq \text{lgu-suffix}$  **then**

lgc-wrong-lgu-suffix (  $n$  ,  $s$  ) **else**

**let**  $s = s$ [suffix'  $\rightarrow t$ ] **in**

**if** lgc-prefix ( lgc-http-prefix ,  $n$  ) **then**

lgc-load-fetch-http ( list-suffix (  $n$  , 5 ) ,  $s$  ) **else**

lgc-load-malformed-lgu (  $n$  ,  $s$  )]

[lgc-trim-newline (  $n$  )  $\equiv$

**let**  $c = n^h$  **in**

**if**  $c = \text{LF}$  **or**  $c = \text{CR}$  **then** lgc-trim-newline (  $n^t$  ) **else**  $n$ ]

## 5.10 Codifying loaded pages

Loading a page involves a first round of processing, loading transitively referenced pages, and a second round of processing. The functions in the following perform the second round of processing.

[lgc-load-codify (  $s$  )  $\equiv$

```
let T :: ((r :: t :: n) :: T) :: T = s['stack'] in
let s = lgc-progress ( 'Codifying' :: n , 3 , s ) in
lgc-exec-events ( s , lgc-load-codify1 ( x , s ) )]
```

Extract the reference  $r$  and type  $t$  of the page to be processed, add a progress message, flush the event queue, and invoke lgc-load-codify1 (  $x$  ,  $s$  ).

[lgc-load-codify1 (  $x$  ,  $s$  )  $\equiv$

```
let T :: ((r :: t :: n) :: s') :: s'' = s['stack'] in
let s = s['stack' → s' :: s''] in
if t = '.lgw' then lgc-load-codify-lgw ( r , s ) else
if t = '.lgr' then lgc-load-codify-lgr ( r , s ) else
lgc-panic ( 'Internal error in lgc-load-codify1' )]
```

Unstack the reference  $r$  and type  $t$  of the page to be processed. Dispatch on  $t$ .

[lgc-load-codify-lgw (  $r$  ,  $s$  )  $\equiv$

```
let s = lgc-load-codify-closure ( r , s ) in
let c = s['cluster'][r] in
let e :: c = lgw-codify ( r , c , s['verbose'] )° in
if e then lgc-proclaim-error ( c , s ) else
let s = s[{'cluster', r} ⇒ c] in
let s = lgc-load-setpath ( s ) in
lgc-load-render ( r , s )]
```

Call lgc-load-codify-closure (  $r$  ,  $s$  ) to add racks of transitively referenced pages to s['cluster'][r] and caches of transitively referenced pages to s['cluster'][r][r]['cluster']. Then use lgw-codify (  $r$  ,  $s$  ,  $v$  ) to codify the page.

[lgc-load-codify-lgr (  $r$  ,  $s$  )  $\equiv$

```
let s = lgc-load-codify-closure ( r , s ) in
let c = s['cluster'][r] in
let c = lgr-cache-restore ( c ) in
let s = s[{'cluster', r} ⇒ c] in
let s = lgc-load-setpath ( s ) in
lgc-load-render ( r , s )]
```

Same as lgc-load-codify-lgw (  $r$  ,  $s$  ) but faster since we have all information except compiled code.

[lgc-load-render (  $r$  ,  $s$  )  $\equiv$

```
let  $s = s[\text{'reference'} \rightarrow r]$  in
let  $e :: p = \text{lgc-render-dirname} ( r , s )^\circ$  in
if  $e$  then lgc-rendering-no-colon (  $s$  ) else
let  $p = p :: \text{'index.html'}$  in
let  $s = \text{lgc-progress} ( \text{'Probing ' } :: p , 4 , s )$  in
let  $s = \text{lgc-push-event} ( s , \text{fileType} ( p ) )$  in
lgc-exec-events (  $s , \text{lgc-load-render1} ( x , s ) )]$ 
```

Probe 'index.html' to see if referenced page has to be re-rendered.

[lgc-load-render1 (  $x$  ,  $s$  )  $\equiv$

```
let  $\langle T , \langle T , f \rangle \rangle = x$  in
let  $s = \text{lgc-progress} ( \text{if } f^h = \text{NULL} \text{ then 'Not found' else 'Found'}$ 
,  $4 , s )$  in
if  $f^h \neq \text{NULL}$  then lgc-load-fetch (  $s$  ) else
let  $r = s[\text{'reference'}]$  in
let  $n = \text{lgc-ref2vt} ( r , s )$  in
let  $s = \text{lgc-progress} ( \text{'Rendering ' } :: n , 3 , s )$  in
lgc-exec-events (  $s , \text{lgc-render} ( x , s ) )]$ 
```

Re-render referenced page if needed. Else go on fetching.

[lgc-load-codify-closure (  $r$  ,  $s$  )  $\equiv$

```
let  $c = s[\text{'cluster'}]$  in
let  $c = \text{lgr-cluster-closure} ( r , c )$  in
 $s[\text{'cluster'} \rightarrow c]$ 
```

Supplement  $s[\text{'cluster'}][r]$  with racks and  $s[\text{'cluster'}][r][r][\text{'cluster'}]$  with caches of transitively referenced pages.

## 5.11 Trisecting

The  $\text{lgw-trisect} ( v )$  function splits the list  $v$  of singleton strings in lgw format into bibliography, dictionary, and flat body and returns a cache  $c$  with the following properties:

- $c[0]$  is the reference  $r$  of the page.
- $c[r][\text{'vector'}]$  is the vector  $v$  of the page.
- $c[r][\text{'bibliography'}]$  is the bibliography of the page.
- $c[r][\text{'dictionary'}]$  is the dictionary of the page.
- $c[r][\text{'body'}]$  is the flat (unparsed) body of the page.

[lgw-parse-string (  $v$  )  $\equiv$

```
let  $n :: v = \text{parse-card} ( v )$  in
lgw-parse-string1 (  $n , v , T$  )]
```

Parse the string at the beginning of  $v$  and return  $c::v'$  where  $c$  is the string and  $v'$  is the unparsed part of  $v$ . A string comprises a cardinal  $n$  followed by  $n$  bytes.

```
[lgw-parse-string1 ( n , v , r ) ≐≐
  if n = 0 then vt2vector ( r ) :: v else
  if v ∈ A then • else
  lgw-parse-string1 ( n - 1 , vt , r :: vh )]
```

Accumulate the result of `lgw-parse-string ( v )` in  $r$ .

```
[lgw-parse-bibliography ( v ) ≐≐ lgw-parse-bibliography1 ( T , v )]
```

Parse a bibliography where a bibliography is a list of strings terminated by an empty string.

```
[lgw-parse-bibliography1 ( r , v ) ≐≐
  let b :: v = lgw-parse-string ( v ) in
  if b = ' then reverse ( r ) :: v else
  lgw-parse-bibliography1 ( b + 0 :: r , v )]
```

Accumulate the result of `lgw-parse-bibliography ( v )` in  $r$ .

Using  $b + 0$  instead of  $b$  above forces bibliographic entries to be stored as integers rather than vectors internally. That is done to circumvent a bug in the logiweb-0.1.x compiler.

```
[lgw-parse-dictionary ( v ) ≐≐ lgw-parse-dictionary1 ( T[0→0] , v )]
```

Parse a dictionary where a dictionary is a list  $\langle i_1, a_1, \dots \rangle$  of pairs of cardinals ended by a zero cardinal. Return the dictionary as an array which maps indices  $i_n$  to arities  $a_n$ . It is understood that index 0 maps to arity 0.

```
[lgw-parse-dictionary1 ( r , v ) ≐≐
  let i :: v = parse-card ( v ) in
  if i = 0 then r :: v else
  let a :: v = parse-card ( v ) in
  lgw-parse-dictionary1 ( r[i→a] , v )]
```

Accumulate the result of `lgw-parse-dictionary ( v )` in  $r$ .

```
[lgw-trisect ( V ) ≐≐
  let b :: v = lgw-parse-bibliography ( V ) in
  let r = bh in
  let c = T[0→r] in
  let c = c[⟨r, 'vector'⟩⇒vt2vector ( V )] in
  let c = c[⟨r, 'bibliography'⟩⇒b] in
  let d :: v = lgw-parse-dictionary ( v ) in
  c[⟨r, 'dictionary'⟩⇒d][⟨r, 'body'⟩⇒v]]
```

Trisect the list  $v$  of singleton strings in lgw format into bibliography, dictionary, and flat body, and produce a cache  $c$  containing them.

## 5.12 Codifying

The `lgw-codify ( r , c , v )` function takes a reference `r` and a cache `c` as input and codifies the flat body found in it. The `v` parameter defines the verbosity (2- means silent, 3 means some output, 4+ means all output). The cache `c` is expected to contain the following:

- `c[0]`: The reference of the page, i.e. a copy of `r`
- `c[r][‘bibliography’]`: The bibliography of the page.
- `c[r][‘dictionary’]`: The dictionary of the page.
- `c[r][‘body’]`: The flat body of the page.
- `c[r][‘cluster’][r’]`: The cache of page `r’` provided `r’` is in the transitive irreflexive bibliography of page `r`.
- `c[r’]`: The rack of page `r’` provided `r’` is in the transitive irreflexive bibliography of page `r`.

The `lgw-codify ( r , c , v )` function returns a cache `c’` in which the following branches are changed:

- `c[r][‘body’]`: The body of the page.
- `c[r][‘expansion’]`: The macro expanded body of the page.
- `c[r][‘codex’]`: The codex of the page.
- `c[r][‘code’]`: Compiled versions of value definitions in the codex.
- `c[r][‘diagnose’]`: The diagnose of the page. The diagnose is maptagged and is computed lazily. In other words, verification is not done by `lgw-codify ( r , c , v )`. Rather, verification occurs when evaluation of the diagnose is forced by untagging it. The page is considered correct if the untagged diagnose is `T`.

```
[lgw-codify ( r , c , v ) ≡  
  lgc-print ( 4 ≤ v , Unpack ) .then.  
  let c = lgw-codify-unpack ( r , c ) in  
  lgc-print ( 4 ≤ v , Initialize ) .then.  
  let c = lgw-codify-initialize ( r , c ) in  
  lgw-codify1 ( r , c , v , 1 )]
```

```
[lgw-codify1 ( r , c , v , n ) ≡  
  lgc-print ( 3 ≤ v , lgc-ordinal ( n ) :: reading ) .then.  
  lgc-print ( 4 ≤ v , Compile ) .then.  
  let c = compile ( c ) in  
  lgc-print ( 4 ≤ v , Expand ) .then.]
```

```

let  $t = \text{lgw-codify-expand} ( r , c )$  in
let  $C = c[\langle r, \text{'expansion'} \rangle \Rightarrow t]$  in
let  $C = C[\langle r, \text{'codex'} \rangle \Rightarrow \top]$  in
lgc-print (  $4 \leq v$  , Harvest ) .then.
let  $C = \text{lgw-codify-harvest} ( r , t , c , C )$  in
lgc-print (  $4 \leq v$  , Compare ) .then.
if  $c[r][\text{'codex'}] = C[r][\text{'codex'}]$  then compile (  $C$  ) else
lgw-codify1 (  $r , C , v , n + 1$  ) ]

```

### 5.13 Unpacking

The `lgw-codify-unpack` ( $r$ ,  $c$ ) function parses the flat body  $c[\text{'body'}]$ , stores the result back in  $c[\text{'body'}]$ , and returns the modified  $c$ .

```

[lgw-codify-unpack (  $r , c$  )  $\equiv$ 
  let  $u = \text{lgw-codify-unpacker} ( r , c )$  in
if  $u$  then lgw-codify-unpack1 (  $r , c$  ) else
let  $v = c[r][\text{'body'}]$  in
let  $c = c[\langle r, \text{'body'} \rangle \Rightarrow \top]$  in
let  $t = \text{prune} ( (\text{eval} ( u^3 , \top , c ) )^c \text{ } v^M )^U , c )$  in
 $c[\langle r, \text{'body'} \rangle \Rightarrow t]$  ]

```

```

[lgw-codify-unpacker (  $r , c$  )  $\equiv$ 
  let  $b = c[r][\text{'bibliography'}]^1$  in
if  $b$  then  $\top$  else  $c[b][\text{'codex'}][b][0][0][\text{'unpack'}]$  ]

```

```

[lgw-codify-unpack1 (  $r , c$  )  $\equiv$ 
  let  $v = c[r][\text{'body'}]$  in
let  $a = \text{lgw-make-dict} ( r , c )$  in
let  $t = \text{lgw-parse-tree} ( a , \langle r \rangle , v )$  in
 $c[\langle r, \text{'body'} \rangle \Rightarrow t^h]$  ]

```

Extract the flat body of page  $r$  from the cache  $c$ . Parse it into a tree and store it back as the body of the tree. For the sake of efficiency, the bibliography  $b$  and the length  $l$  of the bibliography are computed once. Debugging information is accumulated in  $d$ .

```

[lgw-make-dict (  $r , c$  )  $\equiv$ 
  let  $b = c[r][\text{'bibliography'}]$  in
let  $l = \text{length} ( b )$  in lgw-make-dict1 (  $0 , b , l , c , \top$  ) ]

```

Convert the bibliography  $b$  which has length  $l$  into an array  $a$  for which  $a[R + l \cdot i] = \langle r, i, n \rangle$  where  $R$  is the relative reference of a symbol (the position of the reference in the bibliography),  $i$  is the index of a symbol,  $r$  is the absolute references of the symbol, and  $n$  is the arity of the symbol.

```

[lgw-make-dict1 (  $R , b , l , c , a$  )  $\equiv$ 
  if  $b \in \mathbf{A}$  then  $a$  else ]

```

**let**  $a = \text{lgw-make-dict2} ( R , b^h , l , c[b^h][\text{'dictionary'}] , a )$  **in**  
 $\text{lgw-make-dict1} ( R + 1 , b^t , l , c , a )]$

$[\text{lgw-make-dict2} ( R , r , l , d , a ) \stackrel{\bullet\bullet}{\equiv}$   
**if**  $d \in \mathbf{A}$  **then**  $a$  **else**  
**if not**  $d^h \in \mathbf{Z}$  **then**  $\text{lgw-make-dict2} ( R , r , l , d^h , \text{lgw-make-dict2}$   
 $( R , r , l , d^t , a ) )$  **else**  
**let**  $i = d^h$  **in**  
 $a[1 + R + l \cdot i \rightarrow \langle r , i , d^t \rangle]]$

Add the symbols in the dictionary  $d$  to the array  $a$  described under  $\text{lgw-make-dict} ( r , c )$ .  $R$  and  $r$  are the relative and absolute reference, respectively, of the page being processed and  $l$  is the length of the bibliography.

$[\text{lgw-parse-tree} ( a , d , v ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $c :: v = \text{parse-card} ( v )$  **in**  
**if**  $c = 0$  **then**  $\text{lgw-parse-tree-string} ( d , v )$  **else**  
**let**  $\langle r , i , n \rangle = a[c]$  **in**  
**let**  $t = \text{lgw-parse-tree*} ( a , n , 0 , d , v , \top )$  **in**  
 $(\langle r , i , d \rangle :: t^h) :: t^t]$

Convert the flat tree  $v$  into a tree.  $d$  must be debugging information to be installed in the tree.  $a$  must be the value of  $\text{lgw-make-dict} ( r , c )$  where  $c$  is the cache of the page.

$[\text{lgw-parse-tree*} ( a , n , i , d , v , r ) \stackrel{\bullet\bullet}{\equiv}$   
**if**  $i \geq n$  **then**  $\text{reverse} ( r ) :: v$  **else**  
**let**  $t :: v = \text{lgw-parse-tree} ( a , i :: d , v )$  **in**  
 $\text{lgw-parse-tree*} ( a , n , i + 1 , d , v , t :: r )]$

Parse  $n$  trees in the vector  $v$ .  $d$  and  $a$  are described under the previous function.  $i$  counts from 0 to  $n - 1$ . The list of trees is accumulated in  $t$  in reverse order.

$[\text{lgw-parse-tree-string} ( d , v ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $s :: v = \text{lgw-parse-string} ( v )$  **in**  
 $\langle \langle 0 , s , d \rangle \rangle :: v]$

## 5.14 Initializing

The  $\text{lgw-codify-initialize} ( r , c )$  function stores an initial codex in  $c[r][\text{'codex'}]$  and returns the modified cache  $c$ .

$[\text{lgw-codify-initialize} ( r , c ) \stackrel{\bullet\bullet}{\equiv}$   
**if**  $c[r][\text{'bibliography'}]^t \in \mathbf{P}$  **then**  $c$  **else**  
 $c[\langle r , \text{'codex'} , r , 1 , 0 , \text{'definition'} \rangle \Rightarrow \langle \langle 0 , \text{'proclaim'} \rangle \rangle]]]$

## 5.15 Macro expanding

The `lgw-codify-expand ( r , c )` function macro expands `c[r][‘body’]` and returns the expansion.

```
[lgw-codify-expand ( r , c ) ≡
  let d = c[r][‘codex’][r][0][0][‘macro’] in
  if d ∈ P then lgw-codify-expand1 ( d , c ) else
  let b = c[r][‘bibliography’]1 in if b then c[r][‘body’] else
  let d = c[b][‘codex’][b][0][0][‘macro’] in
  if d ∈ P then lgw-codify-expand1 ( d , c ) else c[r][‘body’]]
```

Macro expand page  $r$  in cache  $c$ . First look up the macro expander of the page itself and call it  $d$ . If found, apply it. Else look up the macro expander of the bed page  $b$  and call it  $d$ . If found, apply it. Else, default to the identity macro expander and return the body of the page.

```
[lgw-codify-expand1 ( d , c ) ≡
  let f = eval ( d3 , T , c ) in
  let t = ( f ” cM )U in
  prune ( t , c )]
```

Evaluate right hand side of the macro expander definition  $d$ . Then apply the result to the cache  $c$  and prune the outcome.

## 5.16 Harvesting

The `lgw-codify-harvest ( r , t , c , C )` function harvests the term  $t$  from page  $r$  in cache  $c$  and accumulate the result in the new cache  $C$  which is eventually returned.

```
[lgw-codify-harvest ( r , t , c , C ) ≡
  let d = c[t][‘codex’][t][t][0][‘definition’] in
  if d then lgw-codify-harvest* ( r , tt , c , C ) else
  if di = ‘hide’ then C else
  if di = ‘proclaim’ then lgw-codify-proclaim ( r , t , C ) else
  if ( di = ‘define’ or di = ‘introduce’ ) then lgw-codify-define ( r , t
  , c , C ) else C]
```

Harvest the term  $t$  from page  $r$  in cache  $c$ . Accumulate the result in the new cache  $C$ . In the first line, store the definition aspect of the root of the tree  $t$  in  $d$ . If  $d$  is empty then recurse.

```
[lgw-codify-harvest* ( r , t , c , C ) ≡
  if t ∈ A then C else lgw-codify-harvest* ( r , tt , c , lgw-codify-harves
  ( r , th , c , C ) )]
```

```
[lgw-proclaim-array ≡ T[
  ‘apply’→2::‘value’][
```

```

‘lambda’→2::‘value’[[
‘true’→0::‘value’[[
‘if’→3::‘value’[[
‘quote’→1::‘value’[[
‘proclaim’→2::‘definition’[[
‘define’→3::‘definition’[[
‘introduce’→3::‘definition’[[
‘hide’→T::‘definition’]]

```

[lgw-codify-proclaim (  $r$  ,  $t$  ,  $C$  ) ≡

```

if  $t^{2r} \neq 0$  then  $C$  else
if  $t^{1r} \neq r$  then  $t^\bullet$  else
let  $d = \text{lgw-proclaim-array}[t^{2i}]$  in
if  $d$  or not  $d^h$  and  $d^h \neq \text{length} ( t^{1t} )$  then
 $t^\bullet$  else
if  $t^{2r} \neq 0$  then  $t^\bullet$  else
 $C[\langle r, \text{'codex'}, r, t^{1i}, 0, d^t \rangle \Rightarrow \langle \langle 0, t^{2i} \rangle \rangle]$ 

```

Add the proclamation  $t$  from page  $r$  to the cache  $C$ .

[lgw-tree2aspect (  $t$  ,  $c$  ) ≡

```

if  $t^r = 0$  then  $t$  else  $c[t^r][\text{'codex'}][t^r][t^i][0][\text{'message'}]^3]$ 

```

Convert the term  $t$  into the aspect represented by the term according to the cache  $c$ . If  $t$  is a string, then  $t$  is itself that aspect. Otherwise, the aspect represented by  $t$  is looked up in  $c$ . If  $t$  does not represent any aspect the  $T$  is returned.

[lgw-codify-define (  $r$  ,  $t$  ,  $c$  ,  $C$  ) ≡

```

let  $a = \text{lgw-tree2aspect} ( t^1 , c )$  in
if  $a$  then  $C$  else
 $C[\langle r, \text{'codex'}, t^{2r}, t^{2i}, a^r, a^i \rangle \Rightarrow t]$ 

```

Find the aspect of the definition  $t$  from page  $r$  in the cache  $c$  and then accumulate the definition in  $C$ .

## 6 Grammars

The Logiweb parser  $\text{lgc-parse} ( g , a )$  converts a list  $a$  of tokens to a parse tree as specified by the grammar  $g$ . We describe tokens and grammars in more detail in the following sections.

### 6.1 Tokens

A Logiweb token can be one of the following:

- A byte
- A string

- A binary include
- A text include
- An `""S` escape sequence
- An `""C` escape sequence
- An `""N` escape sequence

Bytes from the source text are represented as singleton strings which in turn are represented by integers. As an example, a capital A in the input is represented by the singleton string 'A' which in turn is represented by the integer  $65 + 256 = 321$  where 65 is the Unicode UTF-8 representation of a capital A. Hence, a capital A in the input is represented by one token and that token has value 321.

As another example, a capital Æ (Unicode C6) is represented by a C3 (195) byte followed by a 86 (134) byte in UTF-8. Thus, a capital Æ is represented by two tokens, namely a  $195 + 256 = 451$  token followed by a  $134 + 256 = 390$  token.

Note that in the Logiweb programming language, a single character is represented by one or more tokens. This is opposite to languages like C where a token represents one or more characters.

A string is represented as a pair `lgc-esc--::S` where *S* is the string. As an example, `"ABC"` in the source text is represented by a token with value `lgc-esc--::'ABC'`. Recall that the value of `lgc-esc--` is 45 where 45 is the Unicode of a hyphen.

A binary include is represented as the pair `lgc-esc-#::S` where *S* is the name of the file to be included. Such a binary include represents a string whose bytes are taken verbatim from the included file. As an example, `"#ABC"` in the source text is represented by a token with value `lgc-esc-#::'ABC'`. That token represents a string whose bytes are taken from the file named ABC.

A text include is represented as the pair `lgc-esc-$::S` where *S* is the name of the file to be included. Such a text include represents a string whose bytes are taken from the included file except that carriage return and line feed characters are parsed as in Section ???. Note that in order to ensure cross platform interoperability, Logiweb marks the end of a line by a line feed character regardless of what the underlying operating system does.

The escape sequences `""S` is represented by a token with value `lgc-esc-S` which in turn equals the Unicode of the character S. An `lgc-esc-S` token represents the entire source text represented as a string. This corresponds to a binary include of the source file.

The escape sequences `""C` is represented by a token with value `lgc-esc-C` which in turn equals the Unicode of the character C. An `lgc-esc-C` token represents a list of charge definitions.

The escape sequences `""N` is represented by a token with value `lgc-esc-N` which in turn equals the Unicode of the character N. An `lgc-esc-N` token represents a list of name definitions.

## 6.2 Source grammars

In the Logiweb programming language, grammars are defined in the source text. During lexical analysis, the grammar is extracted from the source file and converted into a *trie* structure. A trie structure is essentially defined recursively as an array which maps integers to trie structures.

Hence, grammars exist in two forms: externally in the source text and internally as trie structures. We shall refer to these two forms as source and trie grammars, respectively.

A source grammar may look thus:

```
""D 0
x
y
z
""D 4
" + "
```

The source grammar above defines four constructs,  $x$ ,  $y$ ,  $z$ , and  $\dots + \dots$ .

Furthermore, a source grammar assigns an *charge* (c.f. Section 6.3) to each construct. The source grammar above assigns charge 0 to  $x$ ,  $y$ , and  $z$  and charge 4 to  $\dots + \dots$ .

## 6.3 Associativities

A charge consists of a sequence of integers separated by periods. As an example, 7.9.13 is a charge.

Trailing zeros are considered insignificant so 1.2, 1.2.0, and 1.2.0.0.0 represent the same charge. On the other hand, 1.1 and 1.10 are considered different since the 0 in 1.10 is part of the integer 10.

Associativities are ordered lexically. As an example, we have

$$1 < 1.5 < 2.-10 < 2.-9 < 2.-1 < 2 < 2.1 < 2.9 < 2.10 < 3$$

Constructs with low charge bind tighter than constructs with high charge (so constructs with low charge has high priority and vice versa).

A charge is said to be *even* (*odd*) if its last, non-zero number is even (odd). As examples, 1.2.-10 and 1.2.-10.0 are even and 1.2.-11 and 1.2.-11.0 are odd. As a special case, charge 0 is considered to be even.

Constructs with even charge are *preassociative*. A preassociative construct is left associative in text that runs left to right, right associative in text that runs right to left, top associative in text that runs from top to bottom, counter-clockwise-associative in text written in clock-wise spirals, and so on. Constructs with odd charge are *postassociative*.

## 6.4 Constructs

Each construct has four properties:

- Name
- Charge
- Reference
- Index

A name is a string of characters with the following restrictions:

- A name cannot contain characters with codes below 32 (space).
- A name cannot start or end with a space
- A name cannot have two or more spaces in a row anywhere in the name.
- A name cannot have two or more double quote characters in a row anywhere in the name.
- A name must contain at least one character which is not a double quote character.

In names, double quote characters serve as placeholders. As an example, consider the construct

```
if " then " else "
```

When using that construct, one must replace the three double quote characters with expressions.

As mentioned in Section 6.3, a charge is a sequence of integers separated by periods.

The *reference* of a construct is a cardinal (i.e. a non-negative integer). The reference of a construct is equal to the reference of the *home page* of the construct. The home page of a construct is the page on which the construct is defined.

The *index* of a construct is also a cardinal. Constructs on a page are numbered consecutively, starting with 1. In addition, every page has a *page construct* whose name is the name of the page. The page construct has index 0. The page construct is defined by a `"P` escape sequence.

Each construct also has a number of derived properties:

**arity** The arity of a construct equals the total number of double quote characters in the name of the construct.

**Pre-openness** A construct is pre-open if its name begins with a double quote character and pre-closed otherwise.

**Post-closedness** A construct is post-open if its name ends with a double quote character and post-closed otherwise.

**Fixity** We shall say that a construct is open if it is pre- and post-open and that it is closed if it is pre- and post-closed. We shall say that a construct is prefix if it is pre-closed and post-open and suffix if it is pre-open and post-closed.

Some examples read:

Construct	Pre-openness	Post-openness	Fixity
" + "	Pre-open	Post-open	Open
( " )	Pre-closed	Post-closed	Closed
if " then " else "	Pre-closed	Post-open	Prefix
" factorial	Pre-open	Post-closed	Suffix

Internally, we represent constructs by tuples  $\langle r, i, a, p, n, b, R \rangle$  where

$r$  is the reference of the construct (a cardinal)

$i$  is the index of the construct (a cardinal)

$a$  is the charge of the construct expressed as a list of integers. Charge 0 is represented by the empty list.

$p$  is the post-openness of the construct expressed as a Boolean ( $p' = \top$  means that the construct is post-open).

$n$  is the name of the construct expressed as a list of singleton strings.

$b$  is the binary encoding of the construct used in Logiweb vectors.

$R$  is the relative reference of the construct (the position of the construct in the bibliography).

In Logiweb vectors, constructs are represented by byte sequences. Suppose a page references  $N$  pages (including reference zero). The construct with index  $i$  from relative reference  $R$  is represented by the number  $1 + i + N \cdot R$  expressed as a list of septets. A cardinal is expressed as a list of septets by expressing the cardinal little endian base 128 and then adding 128 to all bytes except the last.

## 6.5 Qualified constructs

When importing constructs from different pages, the problem may arise that distinct constructs accidentally have the same name. To cope with that, the programmer may decide to *qualify* constructs.

A construct is qualified by splicing a qualifier (a string) into its name. A qualifier must satisfy the following:

- A qualifier cannot contain characters with codes below 32 (space).
- A qualifier cannot contain double quote characters.

A qualifier is spliced into the following location:

- If the construct does not start with a double quote then the qualifier is spliced in in front of the first character.
- If the construct starts with a quote followed by a non-space then the qualifier is spliced in between the first and second character.
- If the construct starts with a quote followed by a space followed by a non-quote then the qualifier is spliced in between the second and third character.
- If the construct starts with a quote followed by a space followed by a quote then the space is doubled and the qualifier is spliced in between the two spaces.

Once the qualifier is spliced in, duplicate spaces in the result are reduced to single spaces and leading and trailing spaces are removed. Because of this, leading spaces in qualifiers have effect only on constructs which start with a quote followed by a non-space. In contrast, trailing spaces in qualifiers do have effect on all constructs except those which start with a double quote followed by a space followed by a double quote. Some examples are given in the following table.

Qualifier	Name	Qualified name
␣base␣	if␣"␣then␣"␣else␣"	base␣if␣"␣then␣"␣else␣"
base␣	if␣"␣then␣"␣else␣"	base␣if␣"␣then␣"␣else␣"
␣base	if␣"␣then␣"␣else␣"	baseif␣"␣then␣"␣else␣"
base	if␣"␣then␣"␣else␣"	baseif␣"␣then␣"␣else␣"
␣base␣	"␣+␣"	"␣base+␣"
base␣	"␣+␣"	"␣base+␣"
␣base	"␣+␣"	"␣base+␣"
base	"␣+␣"	"␣base+␣"
␣base␣	",,"	"␣base␣,"
base␣	",,"	"base␣,"
␣base	",,"	"␣base,"
base	",,"	"base,"
␣base␣	"␣"	"␣base␣"
base␣	"␣"	"␣base␣"
␣base	"␣"	"␣base␣"
base	"␣"	"␣base␣"

The  $\text{lgc-splice} ( q , c )$  splices the qualifier  $q$  into the construct  $c$ :

$[\text{lgc-panic} ( x ) \stackrel{\bullet\bullet}{=} \text{trace} ( x ) .\text{then.} \perp]$

$[\text{lgc-splice} ( q , c ) \stackrel{\bullet\bullet}{=} \text{if } c \in \mathbf{A} \text{ then lgc-panic} ( \text{'Internal error 1 in lgc-splice'} ) \text{ else}$

```

if  $c^h \neq \text{QQ}$  then append ( lgc-left-trim (  $q$  ) ,  $c$  ) else
let  $c = c^t$  in
if  $c \in \mathbf{A}$  then lgc-panic ( 'Internal error 2 in lgc-splice' ) else
if  $c^h \neq \text{SP}$  then QQ::append (  $q$  ,  $c$  ) else
let  $c = c^t$  in
if  $c^h \neq \text{QQ}$  then QQ::SP::append ( lgc-left-trim (  $q$  ) ,  $c$  ) else
QQ::SP::append ( lgc-trim (  $q$  ) , SP:: $c$  ]

```

## 6.6 Specifying qualification in source files

When referencing a page, all constructs of that page are *imported* and made available for use. As an example, the following line references a page named `page1` and makes all constructs of that page available:

```
""R page1
```

As another example, the following line references a page named `page2` and qualifies all its constructs by `_qual2_`:

```
""R qual2 " page2
```

One may specify more than one qualifier:

```
""R qual3 " Qual3 " page3
```

If `page3` defines a construct named `"_+_"` then the reference above allows to reference that construct as `"_qual3_+_"` or `"_Qual3_+_"` but not as `"_+_"`.

One may specify an empty qualifier:

```
""R" qual4 " Qual4 " page4
```

If `page4` defines a construct named `"_+_"` then the reference above allows to reference that construct as `"_qual4_+_"` or `"_Qual4_+_"` or `"_+_"`.

When specifying an empty qualifier, the empty qualifier must be first in the list: if one tries to put it later then one is forced to put two double quote characters next to each other, yielding an escape sequence.

The page construct also allows qualification:

```
""P" myqual " Myqual " mypage
```

The line above defines a page construct named `mypage` and qualifies all locally defined constructs by the empty string and `myqual` and `Myqual`. That allows to define a construct named e.g. `"_+_"` and to refer to it is `"_+_"`, `"_myqual_+_"`, and `"_Myqual_+_"`.

In case `mypage` is later on referenced from some other page, then only `"_+_"` will be imported from `mypage` to the referencing page. The referencing will not import constructs named `"_myqual_+_"` and `"_Myqual_+_"`. But the referencing page may specify its own qualifications.

## 6.7 Grammar ambiguity

The Logiweb language allows the user to define ambiguous grammars. Actually, the compiler makes no attempt to decide whether or not grammars are ambiguous. Rather, the compiler parses the source text according to the grammar and complains if it cannot parse the source or if it can parse the source in more than one way. In the latter case, the source is said to be ambiguous.

A grammar can be ambiguous in a particularly visible way: it can contain multiple constructs sharing the same name. As an example, suppose Page1 and Page2 both define a construct named "`␣+␣`" and that they are referenced thus:

```
"R" Qual1" Page1
"R" Qual2" Page2
```

The resulting grammar will know constructs named "`␣+␣`", "`␣Qual1+␣`", and "`␣Qual2+␣`". However, "`␣+␣`" will be the name of two, distinct constructs: one from Page1 and one from Page2. If the programmer writes e.g. `2␣Qual1+␣3␣Qual2+` then the first plus will be from Page1 and the second from Page2. But if the programmer writes e.g. `2␣+␣3` then the programmer will get an error message stating that the source text can be parsed in more than one way.

## 6.8 Trie grammars

Consider the following source grammar:

```
"D 0
abc
abd
"D 4
" + "
```

When the Logiweb compiler reads a grammar like the one above, it constructs a trie grammar  $g$  with the following properties:

$$\begin{aligned} g['a']['b']['c'][0] &= \langle \dots \rangle \\ g['a']['b']['d'][0] &= \langle \dots \rangle \\ g[' " ][' ' ][' + ' ][' ' " ' ] [0] &= \langle \dots \rangle \end{aligned}$$

The value of  $g['a']['b']['c'][0]$  is a list of *grammar nodes*. The list has one node for each construct named `abc`. In a typical situation, only one construct is named `abc` and the list will have only one node. However, as mentioned in Section 6.7, more than one construct can have the same name, in which case the list will have more than one node.

## 6.9 Grammar nodes

Each grammar node has form  $\langle r, i, C, p, n, b \rangle$  which specifies the reference, index, charge, post-openness, name, and byte representation of the construct, c.f. Section 6.4.

When the Logiweb compiler translates a source text, it loads all referenced pages and assembles a grammar comprising all constructs of all referenced pages plus all constructs defined by the page itself. All constructs are qualified as specified in "P and verb+"R+ statements before they enter the grammar.

The following functions may be used for accessing individual fields of grammar nodes.

[lgc-node2ref ( *n* )  $\equiv$   $n^0$ ]

[lgc-node2idx ( *n* )  $\equiv$   $n^1$ ]

[lgc-node2charge ( *n* )  $\equiv$   $n^2$ ]

[lgc-node2open ( *n* )  $\equiv$   $n^3$ ]

[lgc-node2closed ( *n* )  $\equiv$  **not**  $n^3$ ]

[lgc-node2name ( *n* )  $\equiv$   $n^4$ ]

[lgc-node2binary ( *n* )  $\equiv$   $n^5$ ]

[lgc-node2relref ( *n* )  $\equiv$   $n^6$ ]

## 6.10 Grammar construction

The lgc-grammar1 ( *x* , *s* ) function adds the following to the state:

*s*['grammar'] The grammar containing constructs from the page being translated and its directly referenced pages.

*s*['dictionary'][*i*] Association list whose entires have form *i* :: *a* where *i* is the index of a construct of the current page and *a* is its associated arity. Sorted in descending *i*.

*s*['binary'][*i*] The binary representation of the construct with index *i* from the page being translated.

*s*['name'][*i*] The name of the construct with index *i* from the page being translated.

*s*['charge'][*i*] The charge of the construct with index *i* from the page being translated.

*s*['index'] The next unused index.

The *x* parameter allows lgc-grammar1 ( *x* , *s* ) to be invoked using by an exec event, but *x* is ignored as it is not supposed to contain meaningfull events.

[lgc-grammar ( *s* )  $\equiv$   
     lgc-exec-events ( *s* , lgc-grammar1 ( *x* , *s* ) )]

```

[lgc-grammar1 ( x , s ) ≐
  let b = reverse ( s[‘refbib’] ) in
  let s = s[‘refbib’→b] in
  let s = s[‘grammar’→lgc-grammar-init] in
  let s = lgc-grammar-add-bib ( s[‘bib’] , 1 , b , s ) in
  let e :: s = lgc-grammar-add-def ( s )o in
  if e then lgc-report-messages ( s ) else
  let s = lgc-progress ( ‘Parsing’ , 3 , s ) in
  lgc-exec-events ( s , lgc-parse1 ( x , s ) ) ]

```

```

[lgc-grammar-init ≐
  let g = T in
  let g = lgc-grammar-init1 ( lgc-esc- , g ) in
  let g = lgc-grammar-init1 ( lgc-esc-# , g ) in
  let g = lgc-grammar-init1 ( lgc-esc-$ , g ) in
  let g = lgc-grammar-init1 ( lgc-esc-C , g ) in
  let g = lgc-grammar-init1 ( lgc-esc-N , g ) in
  let g = lgc-grammar-init1 ( lgc-esc-S , g ) in g ]

```

```

[lgc-grammar-init1 ( x , g ) ≐
  g[⟨x,0⟩⇒⟨⟨T,T,T,F⟩⟩]]

```

## 6.11 Constructs from referenced pages

```

[lgc-grammar-add-bib ( B , R , b , s ) ≐
  if b ∈ A then s else
  let s = lgc-grammar-add-ref ( Bh , R , bh , s ) in
  lgc-grammar-add-bib ( Bt , R + 1 , bt , s ) ]

```

Add all constructs in referenced pages to the grammar in the state  $s$ .  $b$  is a list of references of pages which have not yet been added.  $B$  is the list of the same references but taken from the source text.  $B$  contains prefixes to be added to constructs.  $R$  is the relative reference of the first element of  $b$ .

```

[lgc-grammar-add-ref ( P , R , r , s ) ≐
  let P = Pt in
  let P = if P then ⟨T⟩ else P in
  let d = s[‘cluster’][r][r][‘dictionary’] in
  lgc-grammar-add-dictionary ( P , R , r , d , s ) ]

```

Add all constructs from the page with reference  $r$  and relative reference  $R$  to the grammar.  $P$  has form  $\langle n, p_1, \dots, p_n \rangle$  where  $n$  is the way the page was referenced in the source text and the  $p$ 's are prefixes to be added to grammatical constructs.

```

[lgc-grammar-add-dictionary ( P , R , r , d , s ) ≐
  if d then s else

```

**if**  $d^h \in \mathbf{Z}$  **then** lgc-grammar-add-construct (  $P, R, r, d, s$  ) **else**  
 lgc-grammar-add-dictionary (  $P, R, r, d^t, \text{lgc-grammar-add-dictionar}$   
 (  $P, R, r, d^h, s$  ) )]

Add all constructs in the dictionary  $d$  to the grammar.  $P$  is the list of prefixes to be added to constructs and  $r$  and  $R$  are the reference and relative reference, respectively, of the page being added.

[lgc-grammar-add-construct (  $P, R, r, d, s$  )  $\bullet\bullet$

**let**  $i :: a = d$  **in**  
**let**  $n = \text{lgc-aritysymboll2vt}$  (  $r, i, a, s$  ) **in**  
**let**  $c = \text{lgc-grammar-get-charge}$  (  $r, i, s$  ) **in**  
 lgc-grammar-add-construct1 (  $P, R, i, c, n, s$  )]

Add the construct in the subdirectory  $d$  to the grammar.  $d$  has form  $i :: a$  where  $i$  and  $a$  are the index and arity, respectively, of the construct. The function looks up the name  $n$  and charge  $c$  of the construct and adds them to the grammar.

[lgc-grammar-add-construct1 (  $P, R, i, c, n, s$  )  $\bullet\bullet$

**if**  $P$  **then**  $s$  **else**  
**let**  $q :: P = P$  **in**  
**let**  $N = \text{lgc-splice}$  (  $q, n$  ) **in**  
**let**  $p = (N \text{ last} = \text{QQ})$  **in**  
**let**  $b = \text{lgc-card2septet}^* ( 1 + R + i \cdot (1 + \text{length} ( s[\text{'bib'}] ) ) )$  **in**  
**let**  $t = \langle R, i, c, p, N, b, R \rangle$  **in**  
**let**  $s = \text{push}^* ( s, \text{'grammar'} :: \text{append} ( N, \langle 0 \rangle ), t )$  **in**  
 lgc-grammar-add-construct1 (  $P, R, i, c, n, s$  )].

Splice the qualifiers in the list  $P$  into the name  $n$ , construct the tuple  $t$  which represents the construct, and add it to the grammar.

[lgc-card2septet\* (  $c$  )  $\bullet\bullet$

**if**  $c < \text{septet-base}$  **then** bt2vector\* (  $c$  ) **else**  
**let**  $c :: r = \text{floor}$  (  $c, \text{septet-base}$  ) **in**  
 bt2vector\* (  $r + \text{septet-base}$  ) :: lgc-card2septet\* (  $c$  )]

Express the cardinal  $c$  as a list of septets.

[lgc-aritysymboll2vt (  $r, i, a, s$  )  $\bullet\bullet$

lgc-aritysymboll2vt1 (  $r, i, a, s[\text{'cluster'}][r]$  )]

Find or construct a name for the symbol with reference  $r$ , index  $i$ , and arity  $a$  using the state  $s$ .

[lgc-aritysymboll2vt1 (  $r, i, a, c$  )  $\bullet\bullet$

**let**  $N = c[r][\text{'codex'}][r][i][0][\text{'name'}]$  **in**  
**if**  $N$  **then** lgc-grammar-default-construct (  $r, i, a$  ) **else**  
**let**  $\langle \mathbb{T}, \mathbb{T}, \mathbb{T}, n \rangle = N$  **in**  
**if**  $n^r \neq 0$  **then** lgc-grammar-default-construct (  $r, i, a$  ) **else**  
**let**  $n = \text{lgc-trim-contract}$  ( vt2vector\* (  $n^i$  ) ) **in**

**if**  $n = \top$  **or**  $n = \langle \text{QQ} \rangle$  **then** lgc-grammar-default-construct (  $r$  ,  $i$  ,  $a$  ) **else**

**if** lgc-grammar-valid-construct (  $a$  ,  $n$  ) **then**  $n$  **else**

lgc-grammar-default-construct (  $r$  ,  $i$  ,  $a$  )]

Find or construct a name for the symbol with reference  $r$ , index  $i$ , and arity  $a$  using the cache  $c$ .

Look up the name definition  $N$ . If no definition is found, use the default name. If a name definition is found, extract the right hand side  $n$  of the definition. If  $n$  is no string, use the default name. Else normalize the string. If the result is one of the two forbidden constructs, use the default name. Finally, if  $n$  has wrong arity or has two double quotes in row, use the default name. Else use the name  $n$ .

[lgc-grammar-valid-construct (  $a$  ,  $n$  )  $\equiv$

**if**  $n$  **then**  $a = 0$  **else**

**let**  $c :: n = n$  **in**

**if**  $c \neq \text{QQ}$  **then** lgc-grammar-valid-construct (  $a$  ,  $n$  ) **else**

$n^h \neq \text{QQ}$  **and** lgc-grammar-valid-construct (  $a - 1$  ,  $n$  )]

Check that the name  $n$  has the right arity and does not have two double quotes in a row.

[lgc-def2charge (  $C$  )  $\equiv$

**if**  $C$  **then**  $\top$  **else**

**let**  $\langle \top, \top, \top, c \rangle = C$  **in**

**if**  $c^f \neq 0$  **then**  $\top$  **else**

**let**  $c = \text{vt2vector}^*( c^i )$  **in**

**let**  $e :: c = \text{lgc-parse-charge1} ( c , \top )^\circ$  **in**

**if**  $e$  **then**  $\top$  **else**  $c$ ]

Convert the charge definition  $C$  to a charge. Return the default charge  $\top$  if the definition is missing or malformed.

[lgc-grammar-get-charge (  $r$  ,  $i$  ,  $s$  )  $\equiv$

lgc-def2charge (  $s$ ['cluster'][ $r$ ][ $r$ ]['codex'][ $r$ ][ $i$ ][0]['charge'] )]

Look up the charge definition  $C$  and convert it to a charge (i.e. a list of integers).

## 6.12 Default names

The lgc-grammar-default-construct (  $r$  ,  $i$  ,  $a$  ) construct returns a default name for the construct with reference  $r$ , index  $i$ , and arity  $a$ .

[lgc-grammar-default-construct (  $r$  ,  $i$  ,  $a$  )  $\equiv$

**let**  $r = \text{lgc-string2mixed} ( \text{vector-subseq} ( r , 1 , 3 ) )$  **in**

**let**  $i = \text{lgc-itoa} ( i )$  **in**

**let**  $a = \text{lgc-grammar-default-arglist} ( a )$  **in**  
 $\text{vt2vector} * ( \langle \# [ ' , r , ' : ' , i , ' ] , a \rangle )$

Construct a default name for the construct with reference  $r$  and index  $i$ .

$[\text{lgc-grammar-default-arglist} ( a ) \stackrel{\bullet\bullet}{\equiv}$   
**if**  $a = 0$  **then**  $\top$  **else**  
 $\langle ( ' , \text{lgc-grammar-default-arglist1} ( a - 1 ) , ' ' ) \rangle$ ]

Construct the arglist of the default name of a construct with arity  $a$ .

$[\text{lgc-grammar-default-arglist1} ( a ) \stackrel{\bullet\bullet}{\equiv}$   
**if**  $a = 0$  **then**  $\top$  **else**  
 $\langle ' ' , ' : ' : \text{lgc-grammar-default-arglist1} ( a - 1 ) \rangle$ ]

Auxiliary function for the previous function.

### 6.13 Constructs from source text

$[\text{lgc-grammar-add-def} ( s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $p = s[\text{'page'}]$  **in**  
**if**  $p$  **then**  $\text{lgc-throw-message} ( s , 0 , \text{'No page name found'} )$  **else**  
**let**  $p :: P = p$  **in**  
**let**  $P = \text{if } P \text{ then } \langle \top \rangle \text{ else } P$  **in**  
**let**  $s = s[\text{'index'} \rightarrow 0]$  **in**  
**let**  $s = \text{lgc-grammar-def-construct} ( P , \top , p , s )$  **in**  
**let**  $d = \text{reverse} ( s[\text{'def'}] )$  **in**  
 $\text{lgc-grammar-add-defs} * ( P , d , s )$ ]

Find the page name  $p$ , the list  $P$  of prefixes, and the list  $d$  of charge sections. Add  $p$  as the construct with index zero and then add all constructs in  $d$ .

$[\text{lgc-grammar-def-construct} ( P , c , n , s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $i = s[\text{'index'}]$  **in**  
**let**  $s = s[\text{'index'} \rightarrow i + 1]$  **in**  
**let**  $a = \text{lgc-arity} ( n )$  **in**  
**let**  $s = \text{push} ( s , \text{'dictionary'} , i :: a )$  **in**  
**let**  $b = \text{lgc-card2septet} * ( 1 + i \cdot ( 1 + \text{length} ( s[\text{'bib'}] ) ) )$  **in**  
**let**  $s = s[\langle \text{'binary'} , i \rangle \Rightarrow b]$  **in**  
**let**  $s = s[\langle \text{'name'} , i \rangle \Rightarrow n]$  **in**  
**let**  $s = s[\langle \text{'charge'} , i \rangle \Rightarrow \text{lgc-charge2vector} * ( c )]$  **in**  
 $\text{lgc-grammar-add-construct1} ( P , 0 , i , c , n , s )$ ]

Add the construct  $n$  with charge  $c$  and prefixes  $P$  to the state  $s$ . The arity of the construct is found by counting double quote characters in  $n$  and the index  $i$  is taken from the state.

[lgc-arity (  $n$  )  $\doteq$   
**if**  $n \in \mathbf{A}$  **then** 0 **else**  
**if**  $n^h \neq \text{QQ}$  **then** lgc-arity (  $n^t$  ) **else**  
 1 + lgc-arity (  $n^t$  )]  
 Count the number of double quote characters in  $n$ .

[lgc-grammar-add-defs\* (  $P$  ,  $d$  ,  $s$  )  $\doteq$   
**if**  $d \in \mathbf{A}$  **then**  $s$  **else**  
**let**  $s = \text{lgc-grammar-add-defs}$  (  $P$  ,  $d^h$  ,  $s$  ) **in**  
 lgc-grammar-add-defs\* (  $P$  ,  $d^t$  ,  $s$  )]  
 Add the list  $d$  of charge sections to the grammar.

[lgc-grammar-add-defs (  $P$  ,  $d$  ,  $s$  )  $\doteq$   
**let**  $p :: d = d$  **in**  
**let**  $p = \text{lgc-parse-charge}$  (  $p$  ,  $s$  ) **in**  
 lgc-grammar-def-construct\* (  $P$  ,  $p$  ,  $d$  ,  $s$  )]  
 Add the charge section  $d$  to the grammar.

[lgc-grammar-def-construct\* (  $P$  ,  $p$  ,  $d$  ,  $s$  )  $\doteq$   
**if**  $d \in \mathbf{A}$  **then**  $s$  **else**  
**let**  $s = \text{lgc-grammar-def-construct}$  (  $P$  ,  $p$  ,  $d^h$  ,  $s$  ) **in**  
 lgc-grammar-def-construct\* (  $P$  ,  $p$  ,  $d^t$  ,  $s$  )]  
 Add all constructs in the list  $d$  of constructs to the grammar.

[lgc-parse-charge (  $a$  ,  $s$  )  $\doteq$   
**let**  $p :: a = a$  **in**  
**let**  $a = \text{lgc-trim-contract}$  (  $a$  ) **in**  
**let**  $e :: c = \text{lgc-parse-charge1}$  (  $a$  ,  $\top$  )<sup>o</sup> **in**  
**if not**  $e$  **then**  $c$  **else**  
**let**  $t = \text{lgc-grammar-get-tuple}$  (  $s$  ,  $a$  ) **in**  
**if**  $t \neq \top$  **and**  $t^t = \top$  **then**  $t^{h^2}$  **else**  
**if**  $t = \top$  **then**  
 lgc-throw-message (  $s$  ,  $p$  , ‘Malformed charge’ ) **else**  
 lgc-throw-message (  $s$  ,  $p$  , ‘Charge refers to ambiguous construct’  
 )]

Remove the position  $p$  from  $a$ . Then parse the singleton string  $a$  into a charge. If that fails, assume  $a$  is a construct and look up the charge of that construct. If no charge is found that way, complain.

[lgc-parse-charge1 (  $a$  ,  $r$  )  $\doteq$   
**let**  $i :: a = \text{lgc-parse-int}$  (  $a$  ) **in**  
**if**  $a \in \mathbf{A}$  **then** reverse ( lgc-parse-charge2 (  $i :: r$  ) ) **else**  
**if**  $a^h \neq \text{'.'}$  **then** • **else**  
 lgc-parse-charge1 (  $a^t$  ,  $i :: r$  )]

Parse the charge  $a$  and accumulate it in reverse order in  $r$ .

[lgc-parse-charge2 (  $r$  )  $\equiv$

**if**  $r^h \neq 0$  **then**  $r$  **else** lgc-parse-charge2 (  $r^t$  )]

Remove leading zeros. Since this is used on a reversed list, this function is effectively used for removing trailing zeros.

[lgc-charge2vector\* (  $c$  )  $\equiv$

**if**  $c \in \mathbf{A}$  **then**  $\langle '0' \rangle$  **else**

append ( lgc-itoa (  $c^h$  ) , lgc-charge2vector\*1 (  $c^t$  ) )]

Convert the charge  $c$  to a singleton list.

[lgc-charge2vector\*1 (  $c$  )  $\equiv$

**if**  $c \in \mathbf{A}$  **then**  $\top$  **else**

$'.'$  :: append ( lgc-itoa (  $c^h$  ) , lgc-charge2vector\*1 (  $c^t$  ) )]

Convert the charge tail  $c$  to a singleton list.

[lgc-grammar-get-tuple (  $s$  ,  $n$  )  $\equiv$

get\* (  $s[\text{'grammar'}]$  ,  $n$  )[0]]

Get the tuple associated to the construct  $n$  in the grammar.

## 7 Parser

As mentioned in the beginning of Section 6, the Logiweb parser lgc-parse (  $g$  ,  $a$  ) converts a list  $a$  of tokens to a parse tree as specified by the grammar  $g$ . We now proceed to define lgc-parse (  $g$  ,  $a$  ) in detail.

### 7.1 Token lists

The list  $a$  of tokens is installed in  $s[\text{'body'}]$  by lexical analysis. Each token in  $a$  has form  $c :: p :: S$ . A token may have one of the following forms:

- $c :: p :: \top$  where  $c$  is a character represented as a singleton string from the source file and  $p$  is the position of that character.
- lgc-esc-- ::  $p :: S$  where  $S$  is a string from the source file and  $p$  is the position of the beginning of the string.
- lgc-esc-# ::  $p :: S$  where  $S$  is the name of a file to be binary included and  $p$  is the position of the beginning of the include directive.
- lgc-esc-\$ ::  $p :: S$  where  $S$  is the name of a file to be text included and  $p$  is the position of the beginning of the include directive.
- lgc-esc-S ::  $p :: \top$  where  $p$  is the position of the ""S escape sequence which gave rise to this token.
- lgc-esc-C ::  $p :: \top$  where  $p$  is the position of the ""S escape sequence which gave rise to this token.

- $\text{lgc-esc-N} :: p :: \text{T}$  where  $p$  is the position of the " $\text{S}$ " escape sequence which gave rise to this token.

## 7.2 Linear parse trees

By a *linear parse tree* we shall mean a text in which understood parentheses have been added.

As an example, consider a grammar with the following productions:

x  
y  
z

\cdot

+

If we use brackets to represent understood parentheses, then the linear parse tree of

$x + y + z$

may read

$[[[x] + [y]] + [z]]$

or

$[[x] + [[y] + [z]]]$

## 7.3 Left parse trees

We now introduce the notion of a *left parse tree*. A linear parse tree can be converted into a left parse tree in three steps:

1. Replace all left brackets that follow a left brace by a left brace.
2. Replace all right brackets which match a left brace by a right brace.
3. Delete all left braces.

As an example,  $[[[x] + [y]] + [z]]$  is converted thus:

$[[[x] + [y]] + [z]]$   
 $\rightarrow \{ \{ x \} + [y] \} + [z]$   
 $\rightarrow \{ \{ x \} + [y] \} + [z]$   
 $\rightarrow \{ x \} + [y] \} + [z]$

The conversion from linear to left parse tree is reversible: The two first steps are obviously reversible, and it is not too hard to see how to reverse the third one.

The goal of  $\text{lgc-parse}(g, a)$  is find a left parse tree  $t$  of the list  $a$  of tokens according to the grammar  $g$ .

## 7.4 Representation of left parse trees

A left parse tree is represented by a list  $t$  of tokens. Each token has form  $c::p::S$  and can have one of the following forms:

- An input token. See Section 7.1 for the seven kinds of input tokens.
- $\text{lgc-esc-left}::p::\top$  a left bracket.
- $\text{lgc-esc-right}::p::v$  a right bracket where  $v$  is the value which represents the construct ended by the bracket.
- $\text{lgc-esc-brace}::p::v$  a right brace where  $v$  is the value which represents the construct ended by the brace.

## 7.5 Return value

If  $\text{lgc-parse}(g, a)$  is unable to parse  $a$  according to the grammar  $g$ , then it returns  $\langle p, s \rangle$  where  $p$  is the farthest position into  $a$  that the parser was able to go before it had to give up.  $s$  is a value useful for constructing an error message.

If  $\text{lgc-parse}(g, a)$  can interpret  $a$  in exactly one way then it returns  $\langle \top, t \rangle$  where  $t$  is the interpretation expressed as a left parse tree.

If  $\text{lgc-parse}(g, a)$  can interpret  $a$  in more than one way then it returns  $\langle t, t' \rangle^\bullet$  where  $t$  and  $t'$  are two, distinct interpretations of  $a$  expressed as two, distinct left parse trees.

## 7.6 Restrictions on left parse trees

The parser parses  $a$  as if all constructs are right associative and have the same charge. Hence,  $x + y + z$  should be interpreted as

$$[[x] + [[y] + [z]]]$$

rather than

$$[[[x] + [y]] + [z]]$$

Or, expressed using left parse trees, it should be interpreted as

$$[x\} + [y\} + [z]]]$$

rather than

$$[x\} + [y]\} + [z]]$$

This right-associative-same-charge parsing is implemented by adding a rule saying that a right brace is not allowed to follow a right bracket.

Furthermore, since we do not permit the empty construct, a right brace or bracket cannot follow a left bracket.

Furthermore, since we do not permit constructs with two double quotes in a row, a left bracket cannot follow a right brace or bracket.

Furthermore, since we do not permit the construct which contains one double quote and nothing else, a right bracket or brace cannot follow a right brace.

Finally, a left bracket cannot follow a left bracket due to the definition of left parse trees.

Hence, left parse trees can have sequences of right brackets, but apart from that, brackets and braces are separated by characters of input.

One consequence of the restrictions on grammars are that every construct contains at least one non-double-quote character. Hence, the parse tree of a page cannot have more nodes than the number of characters in the body of the source text. In principle, this ensures that `lgc-parse ( g , a )` terminates in finite time. In practice, however, `lgc-parse ( g , a )` can take very long time if the user defines a silly grammar. We consider making sensible grammars to be the responsibility of the user. We make absolutely no attempt to help users who define silly grammars.

## 7.7 Functions for invoking charge rules

Charge invocation functions vectorize a left parse tree into its binary representation respecting charge rules. The functions also take care of vectorizing of strings and includes and of autogeneration of name and charge definitions.

In the charge invocation functions, we use parameter names as follows:

<i>s</i>	state
$L = t :: L$	left parse tree, i.e. a list of tokens
$t = \langle c, p, v \rangle$	token
<i>c</i>	character
<i>p</i>	position in source text
<i>v</i>	value in token. The type of <i>v</i> depends on <i>c</i> :
$c = \text{lgc-esc--}$	<i>v</i> is a string
$c = \text{lgc-esc-}\#$	<i>v</i> is the name of a binary include file
$c = \text{lgc-esc-}\$$	<i>v</i> is the name of a text include file
$c = \text{lgc-esc-right}$	<i>v</i> is a list <i>N</i> of nodes
$c = \text{lgc-esc-brace}$	<i>v</i> is a list <i>N</i> of nodes
otherwise	<i>v</i> is T
<i>C</i>	charge
$r = M :: r$	list of marked trees
$M = m :: P$	marked tree
<i>m</i>	mark; normally T, occasionally F
$P = n :: A$	partial tree
$N = n :: N$	list of grammar nodes
$n = \langle R, i, C, p', N', b \rangle$	grammar node
<i>R</i>	relative reference
<i>i</i>	index
<i>p'</i>	post-openness
<i>N'</i>	name of construct
<i>b</i>	byte representation of construct
$A = a :: A$	list of arguments
<i>a</i>	argument as vector tree

## 7.8 Main charge functions

$[\text{lgc-charge} ( L , s ) \stackrel{\bullet\bullet}{\equiv}$   
 $\text{lgc-charge1} ( L , s , \top )]$

Convert the left parse tree *L* into a vector tree *a* which, when flattened, becomes the body of the produced Logiweb vector.

$[\text{lgc-charge1} ( L , s , r ) \stackrel{\bullet\bullet}{\equiv}$   
**if** *L* **then** *r* **else**  
**let**  $t :: L = L$  **in**  
**let**  $r = \text{lgc-charge2} ( t , L , s , r )$  **in**  
 $\text{lgc-charge1} ( L , s , r )]$

Process the left parse tree *L* one token at a time, accumulating the result in *r* as a list of marked, partial trees. As an exception, however, *r* is the final result when *t* is empty. That is due to the way  $\text{lgc-charge2} ( t , L , s , r )$  works.

$[\text{lgc-charge2} ( t , L , s , r ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $\langle c, p, v \rangle = t$  **in**

```

if  $c \geq \text{NULL}$  then  $r$  else
if  $c = \text{lgc-esc-right}$  then  $\text{lgc-charge-right} ( p , v , L , s , r )$  else
if  $c = \text{lgc-esc-brace}$  then  $\text{lgc-charge-brace} ( p , v , L , s , r )$  else
if  $c = \text{lgc-esc-left}$  then  $\text{lgc-charge-left} ( r )$  else
if  $c = \text{lgc-esc--}$  then  $\text{lgc-charge-string} ( v , r )$  else
if  $c = \text{lgc-esc-#}$  then  $\text{lgc-charge-binary} ( v , s , r )$  else
if  $c = \text{lgc-esc-\$}$  then  $\text{lgc-charge-text} ( v , s , r )$  else
if  $c = \text{lgc-esc-S}$  then  $\text{lgc-charge-source} ( s , r )$  else
if  $c = \text{lgc-esc-N}$  then  $\text{lgc-charge-name} ( p , s , r )$  else
if  $c = \text{lgc-esc-C}$  then  $\text{lgc-charge-charge} ( p , s , r )$  else
 $\text{lgc-panic} ( \text{'Internal error in lgc-charge2'} )$ 

```

Destruct the token  $t$  into character  $c$ , position  $p$ , and additional data  $v$ . Then dispatch on  $v$ . All the functions return a list  $r$  of marked, partial trees except  $\text{lgc-charge-left} ( r )$  which returns the final result of charge invocation when it processes the leftmost left bracket.

## 7.9 Charge handling of brackets and braces

```

[ $\text{lgc-charge-right} ( p , N , L , s , r )$ ]  $\equiv$ 
if  $N^t$  then  $\langle T, N^h \rangle :: r$  else
let  $a = \text{lgc-charge-right1} ( L , 0 ) :: N$  in
 $\text{lgc-parse-ambiguous-construct} ( a , s )$ 

```

Process a right bracket. If the construct is ambiguous, throw the position  $p$  and the list  $N$  of possible interpretations. Otherwise, push a marked tree  $M = m :: P$  onto  $r$ . The mark  $m$  is set to  $\top$ . The partial tree  $P = n :: A$  has node  $n = N^h$  and an empty argument list  $A$ . Later on, arguments are pushed onto  $A$ . If the construct is pre-open then the mark  $m$  changes to  $F$  when processing the first argument (which is processed last since arguments are processed in reverse order). For pre-closed constructs, the mark never changes.

```

[ $\text{lgc-charge-right1} ( L , l )$ ]  $\equiv$ 
if  $L \in \mathbf{A}$  then  $0$  else
let  $\langle c, p, v \rangle :: L = L$  in
if  $c = \text{lgc-esc-right}$  then  $\text{lgc-charge-right1} ( L , l + 1 )$  else
if  $c = \text{lgc-esc-brace}$  then  $\text{lgc-charge-right1} ( L , l )$  else
if  $c = \text{lgc-esc-left}$  then  $\text{lgc-charge-right1} ( L , l - 1 )$  else
if  $l > 0$  then  $\text{lgc-charge-right1} ( L , l )$  else  $p$ 

```

Skip  $l$  left brackets, then return the position of the first, proper character.

```

[ $\text{lgc-charge-brace} ( p , N , L , s , r )$ ]  $\equiv$ 
let  $r = \text{lgc-charge-brace0} ( r )$  in
 $\text{lgc-charge-right} ( p , N , L , s , r )$ 

```

Process a right brace. First reorganize the stack according to charge. Then, since a right brace represents a right bracket, call lgc-charge-right ( $p, N, L, s, r$ ).

```
[lgc-charge-brace0 ( r ) ::=
  let ( T :: P ) :: r = r in
  let n = Ph in
  let C = lgc-node2charge ( n ) in
  let p = lgc-node2closed ( n ) in
  let r' = ⟨ F :: P ⟩ in
  lgc-charge-brace1 ( C , p , r' , r )]
```

A right brace indicates that the construct at the top of the stack  $r$  is pre-open and that we are about to parse its first argument, so we change the mark of the top element of  $r$  to F. Then we let the construct bubble up the parse tree to its proper location according to its charge. That is done by lgc-charge-brace1 ( $C, p, r', r$ ) which moves the list  $r'$  of marked trees down the stack  $r$  to its proper location according to the charge  $C$ .

```
[lgc-charge-brace1 ( C , p , r prime , r ) ::=
  if r then revappend ( r' , r ) else
  let m :: n :: A = rh in
  if p and not m then lgc-charge-brace1 ( C , p , rh :: r' , rt ) else
  if A ≠ T or lgc-node2closed ( n ) then revappend ( r' , r ) else
  if lgc-less-charge ( C , lgc-node2charge ( n ) ) then revappend ( r' , r ) else
  rh :: lgc-charge-brace1 ( C , p , r' , rt )]
```

Bubble the list  $r'$  of marked trees to its proper location in the  $r$ .  $C$  and  $p$  are the charge and post-closedness, respectively, of the last element of  $r'$ . If  $r$  is empty,  $r'$  is already at the root and cannot bubble further. To bubble, the construct at the top of the stack must be post-open and must be in the state where its last argument is being processed. Since arguments are processed in reverse order, the last argument is being processed iff the argument list  $A$  is empty. Finally, the charge  $C$  must be larger than the charge of the top element for  $r'$  to bubble. When  $r'$  bubbles, it pushes the node above in front of it if the last element of  $r'$  is post-closed.

```
[lgc-less-charge ( x , y ) ::=
  let a :: x = x in
  let b :: y = y in
  let a = if a then 0 else a in
  let b = if b then 0 else b in
  if a < b then T else
  if a > b then F else
```

**if  $x$  and  $y$  then oddp (  $a$  ) else**  
 lgc-less-charge (  $x$  ,  $y$  )]

Compare the charges  $x$  and  $y$  and return **T** if  $x$  is less than  $y$ . The ordering is lexicographic with the following modifications:

1. If one of the two is shorter than the other, then the shorter one is padded with zeros to match the length of the longer.
2. If  $x$  and  $y$  are equal after padding, then return **T** if  $x$  and  $y$  end with an odd number. Constructs whose charge end with an odd number are right associative, so they should not bubble when they meet their equal.

[lgc-charge-left (  $r$  )  $\stackrel{\bullet\bullet}{=}$   
**let** (  $T :: n :: A$  ) ::  $r = r$  **in**  
 lgc-charge-left1 (  $n$  ,  $A$  ,  $r$  )]

Process a left bracket. When processing in reverse order, a left bracket closes one right bracket and zero, one or more right braces. The right braces to be closed need not be the ones that match the left bracket since the stack  $r$  has been reorganized according to charge. Rather, we use the mark of the marked trees in  $r$  to find out how many constructs to close.

The construct at the top of the stack always has mark **T** so we unstack the top element, discards the mark, and destructs the partial tree into a node  $n$  and an argument list  $A$ . Then we use lgc-charge-left1 (  $n$  ,  $A$  ,  $r$  ) to process further constructs from  $r$  until  $r$  is empty or the top element has mark **T**. In other words, we process the top element of  $r$  which is known to have mark **T** and then process all constructs at the top of  $r$  which have mark **F**.

[lgc-charge-left1 (  $n$  ,  $A$  ,  $r$  )  $\stackrel{\bullet\bullet}{=}$   
**let**  $a =$  lgc-node2binary (  $n$  ) ::  $A$  **in**  
**if**  $r \in \mathbf{A}$  **then**  $a$  **else**  
**let** (  $m :: n :: A$  ) ::  $r = r$  **in**  
**if**  $m$  **then** (  $T :: n :: a :: A$  ) ::  $r$  **else**  
 lgc-charge-left1 (  $n$  ,  $a :: A$  ,  $r$  )]

When lgc-charge-left1 (  $n$  ,  $A$  ,  $r$  ) is called,  $n$  and  $A$  have just been popped from the top of  $r$ . That top element is a partial tree which has just been completed and, thus, has become a complete tree representing a subterm of the body of the page. We convert that just completed tree to its representation as a vector tree  $a$ . When  $a$  is flattened later on, that becomes the sequence of bytes representing the just completed tree. If  $r$  has become empty we return  $a$  with no further warning. Normally, we return a stack  $r$ , so this could baffle the caller. However, the stack  $r$  becomes empty exactly when the left parse tree  $L$  becomes empty, so lgc-charge1 (  $L$

,  $s$ ,  $r$ ) will know that  $r$  is not a stack but, rather, the final result of charge invocation. If  $r$  is not empty we look at the mark  $m$  of the new top element. If  $m = \top$  then we can close no more constructs and add  $a$  to the argument list of the top element. Otherwise we also add  $a$  to the argument list but then we recurse.

## 7.10 Charge handling of strings

- [lgc-charge-string ( $v$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
 lgc-charge-string1 ( vector-length ( $v$ ),  $v$ ,  $r$  )]  
 Replace the top element of  $r$  by the string  $v$ .  $v$  must be a vector.
- [lgc-charge-string1 ( $l$ ,  $v$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
 lgc-charge-bytes ( lgc-string2bytes ( $l$ ,  $v$ ),  $r$  )]  
 Replace the top element of  $r$  by the string  $v$  of length  $l$ .  $v$  may be an arbitrary vector tree.
- [lgc-string2bytes ( $l$ ,  $v$ )  $\stackrel{\bullet\bullet}{\equiv}$   
 NULL :: lgc-card2septet\* ( $l$ ) ::  $v$ ]  
 Convert the vector tree  $v$  of length  $l$  into the binary representation of a string.
- [lgc-charge-bytes ( $b$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $n = \langle \top, \top, \top, \top, \top, b \rangle$  **in**  
 $\langle \top, n \rangle :: r^\dagger$   
 Replace the top element of  $r$  by the bytes  $b$ .  $b$  may be an arbitrary vector tree.
- [lgc-charge-binary ( $v$ ,  $s$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $v = s[\text{'includes'}][v]$  **in**  
 lgc-charge-string1 ( length ( $v$ ),  $v$ ,  $r$  )]  
 Replace the top element of  $r$  by the include file named  $v$ .
- [lgc-charge-text ( $v$ ,  $s$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $v = s[\text{'includes'}][v]$  **in**  
**let**  $v = \text{lgc-charge-text1} ( v, \top, \top )$  **in**  
 lgc-charge-string1 ( length ( $v$ ),  $v$ ,  $r$  )]  
 Replace the top element of  $r$  by the include file named  $v$  after processing of newlines.
- [lgc-charge-text1 ( $a$ ,  $i$ ,  $r$ )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $a \in \mathbf{A}$  **then** reverse ( $r$ ) **else**  
**let**  $c :: a = a$  **in**  
**if**  $c = i$  **then** lgc-charge-text1 ( $a$ ,  $\top$ ,  $r$ ) **else**  
**if**  $c = \text{CR}$  **then** lgc-charge-text1 ( $a$ , LF, LF ::  $r$ ) **else**



```
[lgc-parse-headline1 ( v , r ) ≡
  if v ∈ A then r::v else
  let c::V = v in
  if '0' ≤ c and c ≤ '9' then lgc-parse-headline1 ( V , c::r ) else
  if 'A' ≤ c and c ≤ 'F' then lgc-parse-headline1 ( V , c::r ) else
  r::v]
```

Remove all hex digits at the beginning of  $v$ .

## 7.11 Autogeneration of name and charge definitions

```
[lgc-charge-name ( p , s , r ) ≡
  lgc-charge-auto ( p , name , s , r )]
```

Add a name definition for each construct in the dictionary.

```
[lgc-charge-charge ( p , s , r ) ≡
  lgc-charge-auto ( p , charge , s , r )]
```

Add a charge definition for each construct in the dictionary.

```
[lgc-charge-auto ( p , n , s , r ) ≡
  let b = lgc-charge-name-find ( p , " then " , T , s ) in
  let b = b::lgc-charge-name-find ( p , 'def " of " as " enddef' , T ,
  s ) in
  let b = b::lgc-charge-name-find ( p , n , n , s ) in
  let v = lgc-charge-name-find ( p , 'var' , T , s ) in
  let e = lgc-charge-name-find ( p , 'end' , T , s ) in
  let d = s['dictionary'] in
  let b = lgc-charge-auto1 ( d , b , v , s[n] , s['binary'] , e ) in
  lgc-charge-bytes ( b , r )]
```

Find the constructs (lgcdef, lgcvar, lgcthen, lgcend, and either lgcname or lgccharge) needed for auto-construction of name or charge definitions. Then generate the definitions and replace the head of  $r$  with the result.

```
[lgc-charge-name-find ( p , n , d , s ) ≡
  let q = vt2vector* ( 'lgc' ) in
  let n = lgc-splice ( q , vt2vector* ( n ) ) in
  let b = lgc-charge-name2binary ( lgc-splice ( q , n ) , s ) in
  if b ≠ T then b else
  let b = lgc-charge-name2binary ( n , s ) in
  if b ≠ T then b else
  if d ≠ T then lgc-string2bytes ( vector-length ( d ) , d ) else
  let m = 'Cannot generate name and charge definitions ("N and
  "C)') in
  let m = m::LF::'Can find neither '::n::' nor '::lgc-splice ( q ,
  n ) in
  lgc-throw-message ( s , p , m )]
```

Convert construct  $n$  prefixed with `lgclgc` or `lgc` to binary, defaulting to the string  $d$ , if provided.

```
[lgc-charge-name2binary ( n , s ) ≡
  let N = lgc-grammar-get-tuple ( s , n ) in
  let n = lgc-charge-best-node ( N ) in
  lgc-node2binary ( n )]
```

Convert the name  $n$  given as a vector tree to a vector tree of bytes. Return `T` if  $n$  is not found.

```
[lgc-charge-best-node ( N ) ≡
  if N ∈ A then T else
  let n :: N = N in
  let n' = lgc-charge-best-node ( N ) in
  if n' then n else
  let R = lgc-node2relref ( n ) in
  let R' = lgc-node2relref ( n' ) in
  if R < R' then n else
  if R' < R then n' else
  let i = lgc-node2idx ( n ) in
  let i' = lgc-node2idx ( n' ) in
  if i < i' then n else n']
```

Return the node with lowest relative reference and index from the list  $N$  of nodes. If  $N$  is empty return `T`.

```
[lgc-charge-auto1 ( d , b , v , A , B , r ) ≡
  if d ∈ A then r else
  let ( i :: a ) :: d = d in
  let S = A[i] in
  let S = lgc-string2bytes ( length ( S ) , S ) in
  let r = b :: B[i] :: repeat ( a , v ) :: S :: r in
  lgc-charge-auto1 ( d , b , v , A , B , r )]
```

Generate name or charge definitions for all constructs in the dictionary  $d$ .  $b$  must contain the binary representations of `lgcthen`, `lgcdef`, and `lgcname/lgccharge` in that order.  $v$  must contain the binary representation of `lgcvar`.  $A$  must be an array from indexes to right hand sides (i.e. names or charges) expressed as singleton lists.  $B$  must be an array from indexes to binary representations of constructs.

## 7.12 Vectorizing

```
[lgc-vectorize ( L , s ) ≡
  let v = lgc-charge ( L , s ) in
  let v = lgc-add-dict ( s['dictionary'] , v ) in
  let v = lgc-add-bib ( s['refbib'] , v ) in
```

```
let v = lgc-add-ref ( s , v ) in
v]
```

Reorganize the left parse tree  $L$  according to charge and add dictionary and bibliography to form the vector of the page being translated.

```
[lgc-add-dict ( d , v ) ≡
  if d ∈ A then NULL::v else
  let (i::a)::d = d in
  let v = lgc-add-dict ( d , v ) in
  if i = 0 then v else
  let i = lgc-card2septet* ( i ) in
  let a = lgc-card2septet* ( a ) in i::a::v]
```

Add the dictionary  $d$  to the vector tree  $v$ . The dictionary  $d$  is supposed to have form  $\langle i::a, \dots \rangle$ . The dictionary is supposed to be sorted in descending  $i$  but we do not depend on that. The last element (the page symbol) is supposed to have form  $0::0$  but we do not depend on that either.

```
[lgc-add-bib ( b , v ) ≡
  if b ∈ A then NULL::v else
  let r::b = b in
  let r = lgc-ref2vector* ( r ) in
  let v = lgc-add-bib ( b , v ) in r::v]
```

Add the bibliography  $b$  to the vector tree  $v$ .

```
[lgc-ref2vector* ( r ) ≡
  let l = vector-length ( r ) in
  let l = lgc-card2septet* ( l ) in
  let r = vt2vector* ( r ) in
  l::r]
```

Convert the reference  $r$  to a vector tree suited for inclusion in the vector of the page being translated.

```
[lgc-ref-version ≡ bt2vector ( 1 )]
```

```
[lgc-hex2card ( c ) ≡
  if '0' ≤ c and c ≤ '9' then c - 0 else
  if 'A' ≤ c and c ≤ 'F' then c - A + Base else T]
```

Convert the singleton string  $c$  to its hex value (or  $T$  if  $c$  is not a hex digit).

```
[lgc-mixed2vector* ( R ) ≡
  if Rt ∈ A then T else
  let c::d::R = R in
  let D = bt2vector ( lgc-hex2card ( c ) · 16 + lgc-hex2card ( d ) ) in
  D::lgc-mixed2vector* ( R )]
```

```

[lgc-add-ref ( s , v ) ≡
  let h :: H :: T = lgc-parse-headline ( s[‘source’] ) in
  if h then lgc-add-ref1 ( s , v ) else
  let H = lgc-mixed2vector* ( reverse ( H ) ) in
  let t = list-suffix ( H , 21 ) in
  let R = ripemd ( t :: v ) in
  let r = vt2vector* ( lgc-ref-version :: R :: t ) in
  if H ≠ r then lgc-add-ref1 ( s , v ) else
  let r = append ( lgc-card2septet* ( length ( r ) ) , r ) in
  r :: v]

```

```

[lgc-add-ref1 ( s , v ) ≡
  let ⟨m, e⟩ = s[‘time’] in
  let m = lgc-card2septet* ( m ) in
  let e = lgc-card2septet* ( e ) in
  let R = ripemd ( m :: e :: v ) in
  let r = vt2vector* ( lgc-ref-version :: R :: m :: e ) in
  let r = append ( lgc-card2septet* ( length ( r ) ) , r ) in
  r :: v]

```

### 7.13 Invokation of the parser

```

[lgc-parse1 ( x , s ) ≡
  let g = s[‘grammar’] in
  let a = s[‘body’] in
  let e :: v = lgc-parse ( g , a )o in
  if e then lgc-parse-ambiguous ( v , s ) else
  if vh ≠ T then lgc-parse-no-interpretations ( v , s ) else
  let e :: b = vt2vector* ( lgc-vectorize ( vt , s ) )o in
  if e then lgc-report-messages ( b ) else
  let s = s[‘vector’→b] in
  lgc-parse2 ( s )]

```

Parse body according to given grammar. Store the resulting vector in  $s[‘vector’]$ . Then pass control to  $lgc-parse2 ( s )$  to handle header.

```

[lgc-parse2 ( s ) ≡
  let r :: T = lgw-parse-string ( s[‘vector’] ) in
  let s = s[‘reference’→r] in
  let h :: R :: S = lgc-parse-headline ( s[‘source’] ) in
  if h then lgc-parse3 ( s ) else
  let r' = lgc-string2mixed ( r ) in
  if r' = reverse ( R ) then
  lgc-parse4 ( r' , s ) else
  let S = h :: r' :: S in
  let s = lgc-progress ( ‘Writing header back to source’ , 3 , s ) in

```

**let**  $s = \text{lgc-push-event} ( s , \text{fileWrite} ( s[\text{'sourcename'}] , S ) )$  **in**  
 $\text{lgc-parse3} ( s )$ ]

If source has no header, invoke codifier by a call to  $\text{lgc-parse3} ( s )$ . If source has correct header, see if the page is already codified by a call to  $\text{lgc-parse4} ( r' , s )$ . If source has incorrect header, write correct header back to source and invoke codifier.

$[\text{lgc-parse3} ( s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $s = \text{lgc-progress} ( \text{'Codifying'} , 3 , s )$  **in**  
 $\text{lgc-exec-events} ( s , \text{lgc-parse-codify-lgw} ( x , s ) )]$   
 Invoke codifier.

$[\text{lgc-parse4} ( r , s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $s = s[\text{'path'} \rightarrow s[\text{'parameters'}][\text{'path'}]]$  **in**  
 $\text{lgc-parse5} ( s[\text{'mixed'} \rightarrow r] )]$   
 Set  $s[\text{'path'}]$  to the search path and  $s[\text{'mixed'}]$  to the mixed endian hexadecimal representation of the reference. Then see if the page is already codified.

$[\text{lgc-parse5} ( s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $P = s[\text{'path'}]$  **in**  
**if**  $P \in \mathbf{A}$  **then**  $\text{lgc-parse3} ( s )$  **else**  
**let**  $p :: P = P$  **in**  
**let**  $s = s[\text{'path'} \rightarrow P]$  **in**  
**let**  $e :: p = \text{lgc-replace-colon} ( \text{vt2vector*} ( p ) , s[\text{'mixed'}] )^\circ$  **in**  
**if**  $e$  **then**  $\text{lgc-load-no-colon} ( s )$  **else**  
**if**  $\text{lgc-file-suffix} ( p ) \neq \text{lgr-suffix}$  **then**  $\text{lgc-parse5} ( s )$  **else**  
**if**  $\text{lgc-prefix} ( \text{lgc-http-prefix} , p )$  **then**  $\text{lgc-parse5} ( s )$  **else**  
**if**  $\text{lgc-prefix} ( \text{lgc-lgw-prefix} , p )$  **then**  $\text{lgc-parse5} ( s )$  **else**  
**if**  $\text{lgc-prefix} ( \text{lgc-name-prefix} , p )$  **then**  $\text{lgc-parse5} ( s )$  **else**  
**let**  $p =$  **if**  $\text{lgc-prefix} ( \text{lgc-file-prefix} , p )$  **then**  $\text{list-suffix} ( p , 5 )$   
**) else**  $p$  **in**  
**let**  $p = \text{lgc-tilde-expand1} ( p , s )$  **in**  
**let**  $s = \text{lgc-progress} ( \text{'Reading file:'} :: p , 4 , s )$  **in**  
**let**  $s = \text{lgc-push-event} ( s , \text{fileTypeRead} ( p ) )$  **in**  
 $\text{lgc-exec-events} ( s , \text{lgc-parse6} ( x , s ) )]$   
 See if the page is already codified.

$[\text{lgc-parse6} ( x , s ) \stackrel{\bullet\bullet}{\equiv}$   
**let**  $\langle \mathbb{T}, \langle \mathbb{T}, \mathbb{T} :: x \rangle \rangle = x$  **in**  
**if**  $x$  **then**  $\text{lgc-parse5} ( s )$  **else**  
**let**  $e :: c = \text{sl2rack} ( x )^\circ$  **in**  
**if**  $e$  **then**  $\text{lgc-load-malformed-page} ( s )$  **else**  
**let**  $r :: b = c[\text{'bibliography'}]$  **in**  
**let**  $c = \mathbb{T}[0 \rightarrow r][r \rightarrow c]$  **in**  
**let**  $s = s[\langle \text{'cluster'}, r \rangle \Rightarrow c]$  **in**

```

let  $s = \text{lgc-load-codify-closure} ( r , s )$  in
let  $c = s[\text{'cluster'}][r]$  in
let  $c = \text{lgr-cache-restore} ( c )$  in
let  $s = s[\text{'cluster'}, r] \Rightarrow c$  in
let  $s = \text{lgc-progress} ( \text{'Rendering'}, 3 , s )$  in
lgc-exec-events (  $s , \text{lgc-render} ( x , s )$  )

```

## 7.14 Definition of the parser

```

[lgc-parse (  $g , a$  )  $\equiv$ 
let  $t = \langle \langle \text{lgc-esc-left}, 0, \mathbb{T} \rangle \rangle$  in
let  $s = \langle g \rangle$  in
let  $r = \langle 0, s \rangle$  in
lgc-parse-token (  $a , t , s , r , g$  )]

```

Push a left bracket onto  $t$ , set the top of the stack to the entire grammar, and set  $r$  to the best result so far (which is that we have moved zero characters into  $a$ ). Then try to parse a token since a token is the only thing that can follow a left bracket in a left parse tree.

```

[lgc-parse-any (  $p , a , t , s , r , g$  )  $\equiv$ 
let  $r = \text{lgc-parse-left} ( p , a , t , s , r , g )$  in
let  $r = \text{lgc-parse-right} ( p , a , t , s , r , g )$  in
let  $r = \text{lgc-parse-brace} ( p , a , t , s , r , g )$  in
lgc-parse-token (  $a , t , s , r , g$  )]

```

Parse the list  $a$  of tokens, accumulating the associated left parse tree in reverse order in  $t$ .  $p$  is the position in the source file of the last parsed character.  $s$  is a stack of positions in the grammar which indicates where the parser is in the grammar. All elements of  $s$  below the top element are positioned right after a double quote. The value of  $r$  is the best result so far.  $g$  is the entire grammar. During parsing, one may regard  $g$  as a constant.

The ‘any’ function above in turn tries to add a left bracket, a right bracket, a right brace, and a token to  $t$ . This covers all possibilities which is what ‘any’ refers to.

```

[lgc-parse-left (  $p , a , t , s , r , g$  )  $\equiv$ 
if  $t^{\text{hh}} = \text{lgc-esc-left}$  then  $r$  else
let  $G = s^{\text{h}}[\text{QQ}]$  in
if  $G$  then  $r$  else
let  $t = \langle \text{lgc-esc-left}, p, \mathbb{T} \rangle :: t$  in
lgc-parse-token (  $a , t , g :: s^{\text{t}} , r , g$  )]

```

If  $t$  starts with a left bracket we cannot add one more so we return the best result  $r$  found so far. Else, advance the top of the stack  $s$  by a double quote and store the result in  $G$ . If  $G$  is empty, adding

a left bracket to  $t$  is no option and we return the best result  $r$  found so far. Else add a left bracket to  $t$ , replace the top of  $s$  by  $G$ , and then push a fresh copy of the entire grammar  $g$  onto  $s$ .

The function tries to add to  $t$  a *left* bracket which is what ‘left’ in the function name refers to. Only tokens can follow a left bracket, so the function calls the ‘token’ function in case it succeeds to add a left bracket.

```
[lgc-parse-right ( p , a , t , s , r , g ) ••
  let G = sh[0] in
  if G then r else
  let t = ⟨lgc-esc-right, p, G⟩ :: t in
  let s = st in
  if not s then lgc-parse-any ( p , a , t , s , r , g ) else if not
  a then r else
  if rh then ⟨rt, t⟩• else ⊥ :: t]
```

Advance the top of the stack  $s$  by an end of construct marker and store the result in  $G$ . If  $G$  is empty, adding a right bracket to  $t$  is no option and we return the best result  $r$  found so far. Else add a right bracket to  $t$  and pop  $s$ . If  $a$  or  $s$  are non-empty, continue parsing. Else, we have found an interpretation  $t$ . If  $t$  is the second interpretation found, throw an exception containing both interpretations. Else return the interpretation as the best result found so far.

The function tries to add to  $t$  a *right* bracket which is what ‘right’ in the function name refers to.

```
[lgc-parse-brace ( p , a , t , s , r , g ) ••
  let G = sh[0] in
  if G then r else
  if thh = lgc-esc-right then r else
  let t = ⟨lgc-esc-brace, p, G⟩ :: t in
  let G = g[QQ] in
  if G then r else
  let s = st in
  lgc-parse-token ( a , t , G :: s , r , g )]
```

Same as the previous function with the following exceptions: (1) The stack  $s$  is handled differently. (2) To enforce all constructs to be right associative and to have the same charge, we check if  $a^{\text{hh}} = \text{lgc-esc-right}$ . If it is, we cannot add a brace since a right brace is not allowed to follow a right bracket.

The function tries to add to  $t$  a right *brace* which is what ‘brace’ in the function name refers to. Recall that  $t$  may contain right but not left braces. Only tokens can follow a left brace, so the function calls the ‘token’ function in case it succeeds to add a left brace.

```
[lgc-parse-token ( a , t , s , r , g ) ≐
  if a then r else
  let T :: a = a in
  let G = sh[Th] in
  if G then r else
  let ⟨T, p⟩ :: T = a in
  let p = default ( -1 , p ) in
  let t = T :: t in
  let s = G :: st in
  let r = lgc-parse-best ( r , p , s ) in
  lgc-parse-any ( p , a , t , s , r , g )]
```

If  $a$  is empty there are no more tokens to parse and we return the best result  $r$  found so far. Else, we advance the top of  $s$  by the next token and store the result in  $G$ . If  $G$  is empty, moving a token to  $t$  is no option and we return  $r$ . Else, we move the token to  $t$ . The value of the best result found so far is updated if moving the token make us move farther into  $a$  than has been done before.

The function tries to add to  $t$  a *token* which is what ‘token’ in the function name refers to. All four kinds of items (left bracket, right bracket, right brace, or token) can follow a left brace, so the function calls the ‘any’ function in case it succeeds to add a token.

```
[lgc-parse-best ( r , p , s ) ≐
  if rh then r else
  if rh = -1 then r else
  if p = -1 then p :: s else
  if rh > p then r else p :: s]
```

Return the best result of  $r$  and  $p :: s$ . If  $r^h = T$  then  $r$  contains an interpretation which is better than the partial result  $p :: s$ . Otherwise,  $r$  has form  $p' :: s'$  in which case one prefers the larger of  $p$  and  $p'$  except that a value of  $-1$  denotes the end of the file which is larger than any position inside the file.

## 7.15 Message generators

```
[lgc-parse-no-interpretations ( v , s ) ≐
  let p :: S = v in
  let m = ‘Syntax error’ in
  let m = m :: LF :: ‘Cannot parse beyond this point’ in
  let a = lgc-parse-no-inter1 ( S ) in
  let a = vt2vector* ( a ) in
  if a then lgc-error ( s , p , m :: LF :: ‘Expected e.g. end of file’ :: LF :: ‘-
  ) else
  let m = m :: LF :: ‘Expected e.g. |’ in
  let m = m :: list-prefix ( a , 50 ) in
```

**let**  $m = m :: LF :: \text{'---'}$  **in**  
 lgc-error (  $s$  ,  $p$  ,  $m$  )]

Complain about syntactically invalid page. Suggest possible continuation.

[lgc-parse-no-inter1 (  $S$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $S \in \mathbf{A}$  **then**  $\mathbf{T}$  **else**  
 lgc-shortest (  $S^h$  )<sup>t</sup> :: lgc-parse-no-inter1 (  $S^t$  )]

Convert list  $S$  of grammars into possible continuation.

[lgc-shortest (  $g$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $g$  **then**  $1 :: \mathbf{T}$  **else**  
**if**  $g^h \in \mathbf{Z}$  **then** lgc-shortest1 (  $g$  ) **else**  
 lgc-shortest2 (  $g$  )]

Return  $l :: r$  where  $r$  is the shortest production in the grammar  $g$  and  $l$  is the length of  $r$ .

[lgc-shortest1 (  $g$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $c :: g = g$  **in**  
**if**  $c = 0$  **then**  $0 :: \mathbf{T}$  **else**  
**let**  $l :: r = \text{lgc-shortest} ( g )$  **in**  $l + 1 :: c :: r$ ]

Same as lgc-shortest (  $g$  ) except that the head of  $g$  is known to be an integer.

[lgc-shortest2 (  $g$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $r = \text{lgc-shortest} ( g^h )$  **in**  
**let**  $R = \text{lgc-shortest} ( g^t )$  **in**  
**if**  $r$  **then**  $R$  **else**  
**if**  $R$  **then**  $r$  **else**  
**if**  $r^h < R^h$  **then**  $r$  **else**  $R$ ]

Same as lgc-shortest (  $g$  ) except that the head of  $g$  is known not to be an integer.

[lgc-parse-ambiguous (  $v$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$   
 lgc-parse-ambiguous1 ( reverse (  $v^0$  ) , reverse (  $v^1$  ) ,  $s$  )]

Complain about ambiguous page.

[lgc-parse-ambiguous1 (  $a$  ,  $b$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $a^h = b^h$  **then** lgc-parse-ambiguous1 (  $a^t$  ,  $b^t$  ,  $s$  ) **else**  
**let**  $p = \text{default} ( a^{\text{hth}} , b^{\text{hth}} )$  **in**  
**let**  $m = \text{'Ambiguous source text'}$  **in**  
**let**  $m = m :: LF :: \text{lgc-parse-ambiguous2} ( a )$  **in**  
**let**  $m = m :: LF :: \text{lgc-parse-ambiguous2} ( b )$  **in**  
 lgc-error (  $s$  ,  $p$  ,  $m$  )]

Find first difference between the interpretations  $a$  and  $b$  and construct an error message based on that.

[lgc-parse-ambiguous2 (  $a$  )  $\equiv$

**if**  $a$  **then** ‘Could be at the end of the text’ **else**  
**let**  $c = a^{\text{hh}}$  **in**  
**let**  $p = \text{lgc-parse-ambiguous3} ( a )$  **in**  
**let**  $c = \text{lgc-parse-ambiguous4} ( a , 0 )$  **in**  
[‘Could be ’,  $p$ , ‘ of ’,  $c$ ]

Translate the list  $a$  of tokens into a possible interpretation.

[lgc-parse-ambiguous3 (  $a$  )  $\equiv$

**let**  $c = a^{\text{hh}}$  **in**  
**if**  $c = \text{lgc-esc-right}$  **or**  $c = \text{lgc-esc-brace}$  **then** ‘at the end’ **else**  
**if**  $c = \text{lgc-esc-left}$  **then** ‘at the start’ **else** ‘in the middle’]

Find location inside construct (start, middle, or end).

[lgc-parse-ambiguous4 (  $a$  ,  $l$  )  $\equiv$

**if**  $a \in \mathbf{A}$  **then** ‘no construct’ **else**  
**let**  $\langle c, p, v \rangle :: a = a$  **in**  
**if**  $c = \text{lgc-esc-left}$  **then**  $\text{lgc-parse-ambiguous4} ( a , l + 1 )$  **else**  
**if**  $c \neq \text{lgc-esc-right}$  **and**  $c \neq \text{lgc-esc-brace}$  **then**  $\text{lgc-parse-ambiguous4}$   
 $( a , l )$  **else**  
**if**  $l > 0$  **then**  $\text{lgc-parse-ambiguous4} ( a , \text{if } c = \text{lgc-esc-brace} \text{ then } l \text{ else } l - 1 )$  **else**  
**let**  $\langle \langle r, i, C, p, n, b \rangle \rangle = v$  **in**  
**if**  $r$  **then** ‘special construct’ **else** [‘construct ’,  $n$ ]

Skip  $l$  right brackets, then return the name of the next bracket or brace.

[lgc-parse-ambiguous-construct (  $a$  ,  $s$  )  $\equiv$

**let**  $\langle p, \langle R, i \rangle, \langle R', i', \top, \top, n' \rangle \rangle = a$  **in**  
**let**  $m = \text{‘Use of ambiguous construct ’} :: n'$  **in**  
**let**  $m = m :: \text{LF} :: \text{‘Could be construct ’} :: \text{lgc-itoa} ( i ) :: \text{‘ of reference ’} :: \text{lgc-itoa} ( R )$  **in**  
**let**  $m = m :: \text{LF} :: \text{‘Could be construct ’} :: \text{lgc-itoa} ( i' ) :: \text{‘ of reference ’} :: \text{lgc-itoa} ( R' )$  **in**  
 $\text{lgc-throw-message} ( s , p , m )$ ]

[lgc-parse-cannot-trisect (  $s$  )  $\equiv$

$\text{lgc-simple-error} ( \text{‘Could not trisect generated page’} , s )$ ]

[lgc-proclaim-error (  $t$  ,  $s$  )  $\equiv$

**let**  $s = \text{lgc-push-event} ( s , \text{writeln request} ( \text{‘Invalid proclamation:’} ) )$  **in**  
**let**  $s = \text{lgc-push-event} ( s , \text{writeln request} ( \text{lgc-tree2vt} ( t , s ) ) )$  **in**  
 $\text{lgc-do-events} ( s )$ ]

## 7.16 Codification of parsed page

The function defined in this section codifies the page being translated as opposed to the function in Section 5.10 which codifies transitively referenced pages.

```
[lgc-parse-codify-lgw ( x , s ) ≡  
  let v = s[‘vector’] in  
  let e :: c = lgw-trisect ( v )° in  
  if e then lgc-parse-cannot-trisect ( s ) else  
  let r = c[0] in  
  let s = s[‘cluster’, r]⇒c in  
  let s = lgc-load-codify-closure ( r , s ) in  
  let c = s[‘cluster’][r] in  
  let e :: c = lgw-codify ( r , c , s[‘verbose’] )° in  
  if e then lgc-proclaim-error ( c , s ) else  
  let s = s[‘cluster’, r]⇒c in  
  let s = lgc-progress ( ‘Rendering’ , 3 , s ) in  
  lgc-exec-events ( s , lgc-render ( x , s ) )]
```

Trisect and codify the Logiweb vector  $v$ . Then pass control to `lgc-render ( x , s )` to render the page.

# 8 Rendering

## 8.1 Layout of rendering

Pages are rendered in a *rendering directory*. The name of the rendering directory is obtained by replacing the rightmost colon character of the “rendering” option by the reference of the page in mixed endian hexadecimal. The default value of the rendering option is

```
~/logiweb/logiweb/://
```

The rack of a page is stored as “rack.lgr” in the rendering directory.

Rendering of a page results in the following entries in the rendering directory:

**index.html** Overview with html pointers.

**vector.lgw** The vector of the page (‘lgw’ for ‘LoGiWeb’, acknowledging that this format is the main format for exchanges of Logiweb pages over the Internet).

**rack.lgr** The rack of the page.

**ref.lgp** The reference of the page (‘p’ in ‘lgp’ stands for ‘pointer’; the obvious choice ‘r’ for reference was taken by ‘rack’).

**source.lgs** The source text (if known).

**diagnose.txt** Diagnose as text, i.e. as Logiweb source text.

**diagnose.html** Diagnose in html.

**extract.html** Extract (time stamp, bibliography, dictionary, codex, and priority table) in html.

**logiweb.png** Logiweb icon in png.

**logiweb.ico** Logiweb favicon (small icon for browser titlebar).

**page/** Directory containing the user defined rendering of the page.

**page/logiweb.eps** Logiweb icon in eps.

**page/lgwinclude.tex** Include file with useful  $\text{\TeX}$  definitions.

**page/index.html** Typical location of document overview.

**page/page.pdf** Typical location of main document.

**page/diagnose.pdf** Typical location of diagnose in pdf.

**page/bin/** Typical location of generated binaries.

Contents of extract:

- Name and reference of page
- Bibliography containing relref, name, and ref.
- Codex. One codex section per symbol containing: name, index, arity, refname, relref, ref, definitions.
- Priority table.

The following have been abandoned in favour of leaving such representations to user defined rendering: Rack in lisp. Rack in xml. Source translated to html.

## 8.2 State entries

Rendering uses the following entries of the state:

- $s$ ['reference'] The reference as a vector.
- $s$ ['vector'] The page vector as a list of singletons.

## 8.3 Messages

$[lgc-rendering-no-colon ( s ) \ddot{=}$   
lgc-simple-error ( 'Missing colon in rendering option' ,  $s$  )]

## 8.4 Translation of terms to Logiweb source

The construct `lgc-tree2vt ( t , s )` converts the tree  $t$  to a vector tree based on the state  $s$ .

`[lgc-tree2vector* ( t , s ) ≡`

`if t then T else vt2vector* ( lgc-tree2vt ( t , s ) )]`

Convert the tree  $t$  into a source text representation of  $t$ . As a special case, if  $t$  is  $T$  then  $T$  is returned.

`[lgc-string2vt ( t ) ≡`

`if t = " then QQ::QQ::' else`

`let t = vector2vector* ( t ) in`

`(if th = QQ then QQ::QQ::' else QQ)::lgc-string2vt1 ( t )::QQ]`

Convert the string  $t$  given as a vector to a source representation of that string.

`[lgc-string2vt1 ( t ) ≡`

`if t ∈ A then T else`

`let c::t = t in`

`(if c = QQ then QQ::QQ::!' else c)::lgc-string2vt1 ( t )]`

Convert the string  $t$  given as a singleton list to a source representation in which quotes are escaped.

`[lgc-symbol2vt ( r , i , s ) ≡`

`lgc-symbol2vt1 ( r , i , s[‘cluster’][r] )]`

Convert reference  $r$  and index  $i$  into a symbol name using the state  $s$ .

`[lgc-symbol2vt1 ( r , i , c ) ≡`

`let a = c[r][‘dictionary’][i] in`

`lgc-aritysymbol2vt1 ( r , i , a , c )]`

Convert reference  $r$  and index  $i$  into a symbol name using the cache  $c$ .

`[lgc-tree2vt ( t , s ) ≡`

`lgc-tree2vt0 ( F , F , t , s )]`

Convert the tree  $t$  into a source text representation of  $t$ .

`[lgc-tree2vt0 ( p , q , t , s ) ≡`

`let (r, i)::t = t in`

`if r = 0 then lgc-string2vt ( i ) else`

`let n = lgc-symbol2vt ( r , i , s ) in`

`let P = (nh = QQ) in`

`let Q = (n last = QQ) in`

`let t = lgc-tree*2vt ( P , Q , t , s ) in`

`let t = lgc-tree2vt1 ( n , t ) in`

**if  $q$  and  $P$  or  $p$  and  $Q$  then**  
 $\langle \text{QQ}, \text{QQ}, [\cdot, t, \text{QQ}, \text{QQ}, \cdot] \rangle$  **else**  $t$ ]

Convert the tree  $t$  into a source text representation of  $t$ .  $p$  is true if  $t$  is the first argument of a pre-open construct and  $q$  is true if  $t$  is the last argument of a post-open construct.

[lgc-open  $\stackrel{\bullet\bullet}{\equiv}$   $\langle \text{QQ}, \text{QQ}, [\cdot] \rangle$ ]

[lgc-close  $\stackrel{\bullet\bullet}{\equiv}$   $\langle \text{QQ}, \text{QQ}, [\cdot] \rangle$ ]

[lgc-tree\*2vt (  $P$  ,  $Q$  ,  $t$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $t \in \mathbf{A}$  **then**  $\top$  **else**  
lgc-tree2vt0 (  $P$  ,  $Q$  **and**  $t^t$  ,  $t^h$  ,  $s$  ) :: lgc-tree\*2vt (  $F$  ,  $Q$  ,  $t^t$  ,  $s$  )  
)]

Apply lgc-tree2vt (  $t$  ,  $s$  ) to each element of  $t$ .

[lgc-tree2vt1 (  $n$  ,  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $n \in \mathbf{A}$  **then**  $\top$  **else**  
**let**  $c :: n = n$  **in**  
**if**  $c \neq \text{QQ}$  **then**  $c :: \text{lgc-tree2vt1} ( n , t )$  **else**  
 $t^h :: \text{lgc-tree2vt1} ( n , t^t )$ ]

Merge production  $n$  with treelist  $t$ .

[lgc-ref2vt (  $r$  ,  $s$  )  $\stackrel{\bullet\bullet}{\equiv}$   
lgc-tree2vt (  $\langle \langle r, 0 \rangle \rangle$  ,  $s$  )]

Convert the reference  $r$  to the name of the associated page.

## 8.5 Translation of terms to Logiweb source

The construct tree2vt (  $t$  ,  $c$  ) is not used in the lgc compiler but is included here because of its general applicability. It uses the cache  $c$  instead of the state  $s$ .

[tree2vt (  $t$  ,  $c$  )  $\stackrel{\bullet}{\equiv}$   
**let**  $\langle r, i \rangle :: t = t$  **in**  
**if**  $r = 0$  **then** lgc-string2vt (  $i$  ) **else**  
**let**  $n = \text{lgc-symbol2vt1} ( r , i , c )$  **in**  
**let**  $t = \text{tree*2vt} ( t , c )$  **in**  
lgc-tree2vt1 (  $n$  ,  $t$  )]

Convert the tree  $t$  into a source text representation of  $t$ .

[tree\*2vt (  $t$  ,  $c$  )  $\stackrel{\bullet}{\equiv}$   
**if**  $t \in \mathbf{A}$  **then**  $\top$  **else**  
tree2vt (  $t^h$  ,  $c$  ) :: tree\*2vt (  $t^t$  ,  $c$  )]

Apply tree2vt (  $t$  ,  $c$  ) to each element of  $t$ .

## 8.6 General html constructors

```
[lgc-vector*2html ( w ) ≐  
  if w ∈ A then T else  
  let c :: v = w in  
  if c = LF then CRLF :: lgc-vector*2html ( v ) else  
  if c < SP then lgc-vector*2html ( v ) else  
  if c = ' < ' then '&lt;' :: lgc-vector*2html ( v ) else  
  if c = ' > ' then '&gt;' :: lgc-vector*2html ( v ) else  
  if c = '&' then '&amp;' :: lgc-vector*2html ( v ) else  
  if c ≠ QQ then c :: lgc-vector*2html ( v ) else  
  if lgc-prefix ( lgc-open , w ) then lgc-html-open ( w ) else  
  if lgc-prefix ( lgc-close , w ) then lgc-html-close ( w ) else  
  c :: lgc-vector*2html ( v )]
```

Escape special html characters in  $v$ . Translate understood opening and closing brackets by blue brackets.

```
[lgc-html-open ( w ) ≐  
  let w = lgc-vector*2html ( list-suffix ( w , 3 ) ) in  
  ' < font color="blue" > [< /font > ' :: w]
```

Produce blue opening bracket.

```
[lgc-html-close ( w ) ≐  
  let w = lgc-vector*2html ( list-suffix ( w , 3 ) ) in  
  ' < font color="blue" > ] < wbr/ > < /font > ' :: w]
```

Produce blue closing bracket.

```
[lgc-tree2html ( t , s ) ≐  
  lgc-vector*2html ( vt2vector* ( lgc-tree2vt ( t , s ) ) )]
```

Translate the tree  $t$  to Logiweb source and escape special html characters.

```
[lgc-html-begin ( t ) ≐ ' < ' :: t :: ' > ']
```

Enclose  $t$  in angle brackets, forming an opening tag.

```
[lgc-html-end ( t ) ≐ lgc-html-begin ( ' / ' :: t )]
```

Prepend  $t$  by a slash and enclose in angle brackets, forming a closing tag.

```
[lgc-html-tag ( t ) ≐ lgc-html-begin ( t :: ' / ' )]
```

Suffix  $t$  by a slash and enclose in angle brackets, forming a self contained tag.

```
[lgc-html-br ≐ lgc-html-tag ( ' br ' ) :: CRLF]
```

[lgc-html-wrap (  $t$  ,  $b$  )  $\stackrel{\bullet\bullet}{\equiv}$   
lgc-html-begin (  $t$  ) ::  $b$  :: lgc-html-end (  $t$  )]  
Sandwich the body  $b$  between an opening and closing tag.

[lgc-html-title (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  lgc-html-wrap ( 'title' ,  $t$  )]  
Turn the string  $t$  into a title.

[lgc-html-h2 (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  CRLF :: lgc-html-wrap ( 'h2' ,  $t$  )]  
Turn the string  $t$  into headline.

[lgc-html-h3 (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  CRLF :: lgc-html-wrap ( 'h3' ,  $t$  )]  
Turn the string  $t$  into a second level headline.

[lgc-html-h4 (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  CRLF :: lgc-html-wrap ( 'h4' ,  $t$  )]  
Turn the string  $t$  into a third level headline.

[lgc-html-p (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  lgc-html-wrap ( 'p' ,  $t$  )]  
Turn the string  $t$  into a paragraph.

[lgc-html-it (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  lgc-html-wrap ( 'i' ,  $t$  )]  
Display the string  $t$  in italics.

[lgc-html-tt (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  lgc-html-wrap ( 'tt' ,  $t$  )]  
Display the string  $t$  in fixed width font.

[lgc-html-ptt (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  lgc-html-p ( lgc-html-tt (  $t$  ) )]  
Turn the string  $t$  into a paragraph in fixed width font.

[lgc-html-string (  $t$  )  $\stackrel{\bullet\bullet}{\equiv}$  QQ ::  $t$  :: QQ]  
Enclose  $t$  in double quote characters.

[lgc-html-arg (  $k$  ,  $v$  )  $\stackrel{\bullet\bullet}{\equiv}$   
' ' ::  $k$  :: '=' :: lgc-html-string (  $v$  )]  
Construct a keyword/value pair for inclusion in a tag.

[lgc-html-favicon (  $r$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $a$  = lgc-html-arg ( 'rel' ,  $r$  ) **in**  
**let**  $a$  =  $a$  :: lgc-html-arg ( 'href' , 'logiweb.ico' ) **in**  
**let**  $a$  =  $a$  :: lgc-html-arg ( 'type' , 'image/x-icon' ) **in**  
lgc-html-tag ( 'link' ::  $a$  )]  
Construct a Logiweb favicon for inclusion in the head of an html page. The given “rel” type  $r$  may be ‘icon’ or ‘shortcut icon’.

```
[lgc-html-utf8 ≡
  let a = lgc-html-arg ( 'http-equiv' , 'Content-Type' ) in
  let a = a :: lgc-html-arg ( 'content' , 'text/html; charset=UTF-8'
  ) in
  lgc-html-tag ( 'meta' :: a )]
```

```
[lgc-html-head ( t ) ≡
  let a = lgc-html-utf8 in
  let a = a :: CRLF :: lgc-html-title ( t ) in
  let a = a :: CRLF :: lgc-html-favicon ( 'icon' ) in
  let a = a :: CRLF :: lgc-html-favicon ( 'shortcut icon' ) in
  lgc-html-wrap ( 'head' , a )]
```

Construct a head with the given title.

```
[lgc-html-icon ≡
  let a = lgc-html-arg ( 'alt' , 'Logiweb(TM)' ) in
  let a = a :: CRLF :: lgc-html-arg ( 'align' , 'right' ) in
  let a = a :: CRLF :: lgc-html-arg ( 'src' , 'logiweb.png' ) in
  let a = a :: CRLF :: lgc-html-arg ( 'height' , '62' ) in
  let a = a :: CRLF :: lgc-html-arg ( 'width' , '46' ) in
  let a = a :: CRLF :: lgc-html-arg ( 'hspace' , '30' ) in
  lgc-html-tag ( 'img' :: a )]
```

Construct a Logiweb icon for inclusion in the body of an html page.

```
[lgc-html-href ( r , t ) ≡
  let a = lgc-html-begin ( 'a' :: lgc-html-arg ( 'href' , r ) ) in
  a :: t :: lgc-html-end ( 'a' )]
```

Construct an html reference.

```
[lgc-html-name ( r , t ) ≡
  let a = lgc-html-begin ( 'a' :: lgc-html-arg ( 'name' , r ) ) in
  a :: t :: lgc-html-end ( 'a' )]
```

Construct an html anchor.

```
[lgc-html-named-h3 ( r , t ) ≡
  lgc-html-h3 ( lgc-html-name ( r , t ) )]
```

Construct a second level headline which can serve as an anchor.

```
[lgc-html-address ( s ) ≡
  let r = s[reference] in
  let T = lgc-lgt2grdutc2vt ( lgc-ref2lgt ( r ) , s ) in
  let h = 'http://logiweb.eu/logiweb/doc/compiler/index.html' in
  let a = lgc-html-href ( h , 'The Logiweb compiler (lgc)' ) in
  let h = 'http://logiweb.eu/logiweb/doc/misc/time.html' in
  let a = a :: CRLF :: lgc-html-href ( h , T ) in
```

```

let a = lgc-html-wrap ( ‘address’ , a ) in
lgc-html-p ( a )

```

Footer of auto-generated Logiweb pages.

```

[lgc-html-body ( t , b , s ) ≡
let a = lgc-html-icon in
let a = a::CRLF::lgc-html-h2 ( t ) in
let a = a::CRLF::b::CRLF::CRLF::lgc-html-address ( s ) in
lgc-html-wrap ( ‘body’ , a )]

```

Body of auto-generated Logiweb pages.

```

[lgc-html-page ( t , b , s ) ≡
lgc-html-head ( t )::CRLF::lgc-html-body ( t , b , s ) ]
Autogenerated Logiweb page with title t and body b.
```

```

[lgc-html-help ≡
let h = ‘http://logiweb.eu/logiweb/doc/index.html’ in
lgc-html-href ( h , ‘Help’ )]
```

## 8.7 Rendering

```

[lgc-render ( x , s ) ≡
let r = s[‘reference’] in
let t = lgc-ref2lgt ( r ) in
let u = lgc-lgt2grdutc2vt ( t , s ) in
let e::p = lgc-render-dirname ( r , s )o in
if e then lgc-rendering-no-colon ( s ) else
let n = lgc-tree2html ( ⟨⟨r,0⟩⟩ , s ) in
let s = lgc-render-dir ( p , s ) in
let s = lgc-render-link ( p , s ) in
let s = lgc-render-icons ( p , s ) in
let s = lgc-render-index ( p , n , s ) in
let s = lgc-render-extract ( r , p , n , s ) in
let s = lgc-render-vector ( p , s ) in
let s = lgc-render-ref ( p , s ) in
let s = lgc-render-source ( p , s ) in
let s = lgc-progress ( ‘Verifying’ , 3 , s ) in
lgc-exec-events ( s , lgc-render-verify ( x , s ) )]
```

Render standard contents of root directory of page except diagnose.  
Then pass control to verification.

## 8.8 Rendering directory

```

[lgc-render-add-slash ( p ) ≡
if p then ⊤ else
let c::p = p in
```

**if**  $p \neq \top$  **then**  $c :: \text{lgc-render-add-slash } ( p )$  **else**  
**if**  $c = \text{'/'}$  **then**  $\langle c \rangle$  **else**  $\langle c, \text{'/'} \rangle$

Add a slash to the end of the singleton list  $p$  unless  $p$  already has such a slash.

```
[lgc-render-dirname ( r , s ) ≐
  let r = lgc-string2mixed ( r ) in
  let R :: T = s[‘parameters’][‘rendering’] in
  let R = lgc-render-add-slash ( vt2vector* ( R ) ) in
  let p = lgc-replace-colon ( R , r ) in
  lgc-tilde-expand1 ( p , s )]
```

Return the name of the rendering directory.

```
[lgc-render-dir ( p , s ) ≐
  lgc-push-event ( s , fileMkdir ( p :: ‘page/’ ) )]
```

Create the directory  $p$  and its ancestors. Also create a subdirectory named ‘page’ under  $p$ . (Actually, the function creates  $p/\text{page}$  and all its ancestors, including  $p$  itself).

## 8.9 Rendering of links

```
[lgc-lgs-suffix ≐ reverse ( vt2vector* ( .lgs ) )]
```

```
[lgc-page-name ( s ) ≐
  let n = s[‘parameters’][‘source’]h in
  let n = vt2vector* ( n ) in
  let n = reverse ( n ) in
  let e :: n' = lgc-parse-prefix ( lgc-lgs-suffix , n )o in let n = if
  e then  $n$  else  $n'$  in
  lgc-page-name1 ( n , T )]
```

Return the name of the source file with directory names and suffix ‘.lgs’ removed if present. As an example, if the source file is named ‘../foo.lgs’ then ‘f’, ‘o’, ‘o’ is returned.

```
[lgc-page-name1 ( n , r ) ≐
  if  $n \in \mathbf{A}$  then  $r$  else
  let  $c :: n = n$  in
  if  $c = \text{'/'}$  then  $r$  else
  lgc-page-name1 ( n ,  $c :: r$  )]
```

Remove directory names from the reverse path name  $n$  and accumulate the result in  $r$ .

```
[lgc-render-link ( p , s ) ≐
  if  $s[\text{'stack'}] \neq \top$  then  $s$  else
  let  $p = \text{lgc-cwd-expand } ( p , s )$  in
  let  $L = s[\text{'parameters'}][\text{'link'}]$  in
```

```

let  $N = \text{lgc-page-name} ( s )$  in
lgc-render-link1 (  $p$  ,  $N$  ,  $L$  ,  $s$  )]

```

Generate all links to the page root directory.

```

[lgc-render-link1 (  $p$  ,  $N$  ,  $L$  ,  $s$  ) ≡
if  $L \in \mathbf{A}$  then  $s$  else
let  $l :: L = L$  in
let  $e :: n = \text{lgc-replace-colon} ( \text{vt2vector}^* ( l ) , N )^\circ$  in
let  $n = \text{if } e \text{ then } l \text{ else } n$  in
let  $n = \text{lgc-tilde-expand1} ( n , s )$  in
let  $s = \text{lgc-push-event} ( s , \text{fileMkdir} ( n ) )$  in
let  $s = \text{lgc-push-event} ( s , \text{fileRm} ( n ) )$  in
let  $s = \text{lgc-push-event} ( s , \text{fileSymlink} ( p , n ) )$  in
lgc-render-link1 (  $p$  ,  $N$  ,  $L$  ,  $s$  )]

```

Replace the rightmost colon of each element of the link list  $L$  by the name  $N$  and create a link which points to  $p$ . Also create ancestor directories as needed and overwrite the link if it exists already.

## 8.10 Rendering of non-html

```

[lgc-render-vector (  $p$  ,  $s$  ) ≡
let  $E = \text{fileWrite} ( p :: \text{'vector.lgw'} , s[\text{'vector'}] )$  in
lgc-push-event (  $s$  ,  $E$  )]

```

Generate the vector of the page.

```

[lgc-render-ref (  $p$  ,  $s$  ) ≡
let  $r = s[\text{'reference'}]$  in
let  $r = \text{lgc-string2mixed} ( r )$  in
let  $E = \text{fileWrite} ( p :: \text{'ref.lgp'} , r )$  in
lgc-push-event (  $s$  ,  $E$  )]

```

Generate the vector of the page.

```

[lgc-render-source (  $p$  ,  $s$  ) ≡
let  $S = \text{lgc-render-source1} ( s )$  in
let  $E = \text{fileWrite} ( p :: \text{'source.lgs'} , S )$  in
lgc-push-event (  $s$  ,  $E$  )]

```

Write the source text to the rendering directory. The function is prepared for a situation where the source may be unknown.

```

[lgc-render-source1 (  $s$  ) ≡
let  $v = s[\text{'source'}]$  in
if  $v$  then  $\text{'Source not known, sorry.'}$  else
let  $R = \text{lgc-string2mixed} ( s[\text{'reference'}] )$  in
lgc-add-headline (  $s$  ,  $R$  ,  $v$  )]

```

Add reference to headline. Act sensibly if the source is unknown.

```
[lgc-render-icons ( p , s ) ≐
  let s = lgc-push-event ( s , fileWrite ( p :: 'logiweb.png' , lgc-logiweb.png
  ) ) in
  let s = lgc-push-event ( s , fileWrite ( p :: 'logiweb.ico' , lgc-logiweb.ico
  ) ) in
  lgc-push-event ( s , fileWrite ( p :: 'page/logiweb.eps' , lgc-logiweb.eps
  ) )]]
Write icons to the rendering directory.
```

## 8.11 Rendering of index

```
[lgc-render-index ( p , n , s ) ≐
  let t = 'Logiweb main menu of ' :: n in
  let a = lgc-html-h3 ( 'Rendering' ) in
  let a = a :: CRLF :: lgc-html-href ( 'page/page.pdf' , 'Main text'
  ) in
  let a = a :: CRLF :: lgc-html-href ( 'page/index.html' , 'Index' ) in
  let a = a :: CRLF :: lgc-html-href ( 'page/diagnose.pdf' , 'Diagnose'
  ) in
  let a = a :: CRLF :: lgc-html-h3 ( 'Debugging aids' ) in
  let a = a :: CRLF :: lgc-html-href ( 'extract.html' , 'Extract' ) in
  let a = a :: CRLF :: lgc-html-href ( 'source.lgs' , 'Source' ) in
  let a = a :: CRLF :: lgc-html-href ( 'diagnose.html' , 'Diagnose' ) in
  let a = a :: CRLF :: lgc-html-h3 ( 'Documentation' ) in
  let a = a :: CRLF :: lgc-html-help in
  let a = lgc-html-page ( t , a , s ) in
  let E = fileWrite ( p :: 'index.html' , a ) in
  lgc-push-event ( s , E )]]
Write an html index to the rendering directory.
```

## 8.12 Rendering of extract

```
[lgc-render-extract ( r , p , n , s ) ≐
  let t = 'Logiweb extract of ' :: n in
  let a = lgc-html-href ( 'index.html' , 'Up' ) in
  let a = a :: CRLF :: lgc-html-help in
  let a = a :: CRLF :: lgc-render-extract-toc in
  let a = a :: CRLF :: lgc-render-extract-date ( r , s ) in
  let a = a :: CRLF :: lgc-render-bib ( r , s ) in
  let a = a :: CRLF :: lgc-render-def ( r , s ) in
  let a = a :: CRLF :: lgc-render-charge ( r , s ) in
  let a = lgc-html-page ( t , a , s ) in
  let E = fileWrite ( p :: 'extract.html' , a ) in
  lgc-push-event ( s , E )]]
Render the extract of the page.
```

```
[lgc-render-extract-toc ≡
    let a = lgc-html-href ( '#timestamp' , 'Date of publication'
    )::lgc-html-br in
    let a = a :: lgc-html-href ( '#bibliography' , 'Bibliography'
    )::lgc-html-br in
    let a = a :: lgc-html-href ( '#codex' , 'Definitions' )::lgc-ht

    let a = a :: lgc-html-href ( '#charge' , 'Charges' ) in
    let a = lgc-html-p ( a ) in
    lgc-html-h3 ( 'Table of contents' )::CRLF::a]
Render a table of contents for the extract.
```

```
[lgc-render-extract-date ( r , s ) ≡
    let t = lgc-ref2lgt ( r ) in
    let a = lgc-lgt2grdtc2vt ( t , s ) in
    let a = a :: ' (Gregorian Date / Universal Coordinated
    Time)' in
    let a = a :: lgc-html-br in
    let a = a :: lgc-lgt2mjdtaivt ( t ) in
    let a = a :: ' (Modified Julian Day / International Atomic
    Time)' in
    let a = a :: lgc-html-br in
    let a = a :: lgc-lgt2vt ( t ) in
    let a = a :: ' (Logiweb Time)' in
    let a = lgc-html-p ( a ) in
    lgc-html-named-h3 ( 'timestamp' , 'Date of publication'
    )::CRLF::a]
Render date of publication in three formats.
```

### 8.13 Rendering of bibliography

```
[lgc-render-bib ( r , s ) ≡
    let b = s['cluster'][r][r]['bibliography'] in
    let w = length ( lgc-itoa ( length ( b ) - 1 ) ) in
    let a = lgc-html-named-h3 ( 'bibliography' , 'Bibliography'
    ) in
    a :: CRLF::lgc-html-ptt ( lgc-render-bib1 ( 0 , w , b ,
    s ) )]]
Render bibliography.
```

```
[lgc-render-ref-link ( r , a ) ≡
    let h = lgc-string2mixed ( r ) in
    lgc-html-href ( '././logiweb/'::h::/index.html , a )]
Add link to page with reference r.
```

```
[lgc-render-bib1 ( i , w , b , s ) ≡
    let r :: b = b in
```

```

let  $h$  = lgc-string2mixed (  $r$  ) in
let  $a$  = ‘[?::lgc-ctoa (  $i$  ,  $w$  )::‘]’ in
let  $a$  =  $a$ ::lgc-ref2vt (  $r$  ,  $s$  ) in
let  $a$  =  $a$ ::‘(‘:: $h$ ::‘)’ in
let  $a$  = lgc-render-ref-link (  $r$  ,  $a$  ) in
if  $b$  then  $a$  else
 $a$ ::lgc-html-br::lgc-render-bib1 (  $i + 1$  ,  $w$  ,  $b$  ,  $s$  )]
Render each bibliography entry.

```

## 8.14 Rendering of definitions

```

[lgc-render-def (  $r$  ,  $s$  ) ≡
let  $a$  = lgc-html-named-h3 ( ‘codex’ , ‘Definitions’ ) in
let  $R$  =  $s$ [‘cluster’][ $r$ ][ $r$ ] in
let  $b$  =  $R$ [‘bibliography’] in
let  $c$  =  $R$ [‘codex’] in
let  $d$  =  $R$ [‘dictionary’] in
let  $I$  = array-domain (  $d$  ) in
let  $a$  =  $a$ ::CRLF::lgc-render-def-sym* (  $r$  ,  $r$  ,  $I$  ,  $s$  ) in
let  $c$  =  $c$ [ $r$ → $T$ ] in
if  $c$  then  $a$  else
 $a$ ::CRLF::lgc-render-def1 (  $r$  ,  $b$  ,  $c$  ,  $s$  )]

```

Render all definitions: Extract the rack  $R$  from the state  $s$ . Then extract bibliography  $b$ , codex  $c$ , and dictionary  $d$ .

Then render domestic definitions (i.e. definitions on page  $r$  of symbols from page  $r$ ). The rendering of domestic definitions is special because a symbol gets a section even if it has no definitions. This allows the user to see that the symbol exists and to deduce its arity. In that way it is ensured that the information of the dictionary is present.

When a page has been rendered, it is removed from the codex  $c$  so that it is not rendered more than once.

```

[lgc-render-def1 (  $p$  ,  $b$  ,  $c$  ,  $s$  ) ≡
if  $b$  ∈ A then lgc-render-def2 (  $p$  , array-domain (  $c$  ) ,  $c$  ,  $s$  ) else
let  $r$ :: $b$  =  $b$  in
let  $a$  = lgc-render-def-sym* (  $p$  ,  $r$  , array-domain (  $c$  [ $r$ ] ) ,  $s$  ) in
let  $c$  =  $c$ [ $r$ → $T$ ] in
if  $c$  then  $a$  else
 $a$ ::CRLF::lgc-render-def1 (  $p$  ,  $b$  ,  $c$  ,  $s$  )]

```

Render definitions of symbols from directly referenced pages. Definitions are sorted according to the order in which pages are referenced. If (for some silly reason) the bibliography references some page more than once then the definitions of that page are only stated once.

```
[lgc-render-def2 ( p , R , c , s ) ≡
  if R ∈ A then T else
  let r :: R = R in
  let a = lgc-render-def-sym* ( p , r , array-domain ( c
[r] ) , s ) in
  if R then a else
  a :: CRLF :: lgc-render-def2 ( p , R , c , s )]
Render definitions of symbols from transitively referenced
pages which have not yet been rendered.
```

```
[lgc-render-def-sym* ( p , r , I , s ) ≡
  if I ∈ A then T else
  let i :: I = I in
  let a = lgc-render-def-sym ( p , r , i , s ) in
  if I ∈ A then a else
  a :: CRLF :: lgc-render-def-sym* ( p , r , I , s )]
Render the definitions of the symbols with reference r
and index in the list I of indices
```

```
[lgc-render-def-sym ( p , r , i , s ) ≡
  if r = 0 then lgc-render-def-string ( p , i , s ) else
  let a = lgc-symbol2vt ( r , i , s ) in
  let a = lgc-vector*2html ( a ) in
  let a = lgc-html-h4 ( a ) in
  let b = lgc-render-ref-link ( r , lgc-ref2vt ( r , s ) ) in
  let b = 'Index ' :: lgc-itoa ( i ) :: ofpage :: b in
  let a = a :: CRLF :: lgc-html-p ( b ) in
  let d = s['cluster'][p][p]['codex'][r][i] in
  a :: CRLF :: lgc-render-def-sym1 ( d , array-domain ( d
) , s )]
Render the definitions of the symbol with reference r and
index i.
```

```
[lgc-render-def-string ( p , i , s ) ≡
  let a = lgc-vector*2html ( vt2vector* ( i ) ) in
  let a = lgc-html-h4 ( lgc-html-it ( a ) ) in
  let b = 'Index ' :: lgc-itoa ( i ) :: ofthestringpage(pagezero) in

  let a = a :: CRLF :: lgc-html-p ( b ) in
  let d = s['cluster'][p][p]['codex'][0][i] in
  a :: CRLF :: lgc-render-def-sym1 ( d , array-domain ( d
) , s )]
```

Render the definitions of the string  $i$ .

```
[lgc-render-def-sym1 ( D , R , s ) ≐
  if R ∈ A then T else
  let r :: R = R in
  let d = D[r] in
  let a = lgc-render-def-sym2 ( d , array-domain ( d ) , s
  ) in
  if R ∈ A then a else
  a :: CRLF :: lgc-render-def-sym1 ( D , R , s )]
```

Render definitions in  $D$  for the list  $R$  of aspect references.

```
[lgc-render-def-sym2 ( D , I , s ) ≐
  if I ∈ A then T else
  let i :: I = I in
  let a = lgc-html-p ( lgc-render-def-sym3 ( D[i] , s ) ) in
  if I ∈ A then a else
  a :: CRLF :: lgc-render-def-sym2 ( D , I , s )]
```

Render definitions in  $D$  for the list  $I$  of aspect indices.

```
[lgc-render-def-sym3 ( t , s ) ≐
  if tr ≠ 0 then lgc-tree2html ( t , s ) else
  lgc-html-tt ( ‘Proclamed meaning: ’ :: lgc-html-string ( ti
  ) )]
```

## 8.15 Rendering of charges

```
[lgc-render-charge ( r , s ) ≐
  let a = lgc-html-named-h3 ( ‘charge’ , ‘Charges’ ) in
  let b = s[‘cluster’][r][r][‘bibliography’] in
  let C = lgc-collect-charge ( b , s , T ) in
  a :: CRLF :: lgc-render-charge0 ( b , C , s , T )]
```

Construct  $C$  such that  $\text{get}^* ( C , c )^t$  equals  $F$  if some construct has charge  $c$  and such that  $\text{get}^* ( C , c )^h[r][i]$  equals  $F$  if the symbol with reference  $r$  and index  $i$  has charge  $c$ .

```
[lgc-collect-charge ( b , s , C ) ≐
  if b ∈ A then C else
  let r :: b = b in
  let c = s[‘cluster’][r][r][‘codex’][r] in
  let C = lgc-collect-charge1 ( r , c , C ) in
  lgc-collect-charge ( b , s , C )]
```

Collect all charges used on pages listed the bibliography  $b$  and add them to  $C$ .

```
[lgc-collect-charge1 ( r , c , C ) ≐
  if c ∈ A then C else
```

**if**  $c^h \in \mathbf{Z}$  **then** lgc-collect-charge2 (  $r, c^h, c^t, C$  ) **else**  
**let**  $C = \text{lgc-collect-charge1} ( r, c^h, C )$  **in**  
 lgc-collect-charge1 (  $r, c^t, C$  )  
 Collect all charges used by the subcodex  $c$ .

[lgc-collect-charge2 (  $r, i, c, C$  )  $\stackrel{\bullet\bullet}{=}$   
**let**  $c = \text{lgc-def2charge} ( c[0][\text{'charge'}] )$  **in**  
 lgc-collect-charge3 (  $r, i, c, C$  )  
 Add the symbol with reference  $r$ , index  $i$ , and charge  $c$   
 to  $C$ .

[lgc-collect-charge3 (  $r, i, c, C$  )  $\stackrel{\bullet\bullet}{=}$   
**if**  $c \in \mathbf{A}$  **then** lgc-collect-charge4 (  $r, i, C$  ) **else**  
**let**  $C = \text{if } C^t \neq \mathbf{F} \text{ then } C \text{ else } \top[0 \rightarrow C]$  **in**  
 $C[c^h \rightarrow \text{lgc-collect-charge3} ( r, i, c^t, C[c^h] )]$   
 Add the symbol with reference  $r$ , index  $i$ , and charge  $c$   
 to  $C$  by recursion in  $c$ .

[lgc-collect-charge4 (  $r, i, C$  )  $\stackrel{\bullet\bullet}{=}$   
**if**  $C = \top$  **or**  $C^t = \mathbf{F}$  **then**  $C^h[\langle r, i \rangle \Rightarrow \mathbf{F}] :: \mathbf{F}$  **else**  
 $C[0 \rightarrow \text{lgc-collect-charge4} ( r, i, C[0] )]$   
 Add the symbol with reference  $r$  and index  $i$  to  $C$ . Ex-  
 tend the charge by trailing zeros until we have  $C^t = \mathbf{F}$ .

[lgc-render-charge0 (  $b, C, s, c$  )  $\stackrel{\bullet\bullet}{=}$   
**if**  $C^t = \mathbf{F}$  **then** lgc-render-charge2 (  $b, C, s, c$  ) **else**  
**let**  $D = \text{array-domain} ( C )$  **in** lgc-render-charge1 (  $b, C, D, s, c$  )

[lgc-render-charge1 (  $b, C, D, s, c$  )  $\stackrel{\bullet\bullet}{=}$   
**if**  $D \in \mathbf{A}$  **then**  $\top$  **else**  
**let**  $d :: D = D$  **in**  
**let**  $a = \text{lgc-render-charge0} ( b, C[d], s, d :: c )$  **in**  
**if**  $D \in \mathbf{A}$  **then**  $a$  **else**  
 $a :: \text{CRLF} :: \text{lgc-render-charge1} ( b, C, D, s, c )]$   
 Render all charges in  $C$  in increasing order.  $D$  is the list  
 of not yet processed part of the domain of  $C$ .  $c$  accu-  
 mulates the name of the charge.  $b$  is the bibliography of  
 the page and  $s$  is the state.

[lgc-render-charge2 (  $b, C, s, c$  )  $\stackrel{\bullet\bullet}{=}$   
**let**  $C = C^h$  **in**  
**let**  $r :: b = b$  **in**  
**let**  $a = \text{lgc-render-charge-bib} ( b, C, s )$  **in**  
**let**  $D = \text{array-domain} ( C[r] )$  **in**  
**let**  $a = a :: \text{CRLF} :: \text{lgc-render-charge-self} ( r, D, s )$   
**in**  
**let**  $c = \text{lgc-charge2vector*} ( \text{reverse} ( \text{lgc-parse-charge2} ( c ) ) )$  **in**

lgc-html-h4 ( default ( '0' , c ) )::CRLF::lgc-html-ptt  
( a )]

Render charge section for charge  $c$ .

[lgc-render-charge-bib (  $b$  ,  $C$  ,  $s$  )  $\equiv$

**if**  $b \in \mathbf{A}$  **then**  $\top$  **else**

**let**  $r :: b = b$  **in**

**let**  $a = \text{lgc-render-charge-bib} ( b , C , s )$  **in**

**let**  $D = \text{array-domain} ( C[r] )$  **in**

**if**  $D$  **then**  $a$  **else**

$\text{lgc-symbol2vt} ( r , D^h , s ) :: \text{lgc-html-br} :: a]$

For each page listed in  $b$  render the first construct (if any) which occurs in  $C$ .

[lgc-render-charge-self (  $r$  ,  $D$  ,  $s$  )  $\equiv$

**if**  $D \in \mathbf{A}$  **then**  $\top$  **else**

**let**  $i :: D = D$  **in**

**let**  $a = \text{lgc-render-charge-self} ( r , D , s )$  **in**

$\text{lgc-symbol2vt} ( r , i , s ) :: \text{lgc-html-br} :: a]$

Render all constructs in the list  $D$  of indices.

## 8.16 Verification

[lgc-render-verify (  $x$  ,  $s$  )  $\equiv$

**let**  $r = s[\text{'reference'}]$  **in**

**let**  $p = \text{lgc-render-dirname} ( r , s )$  **in**

**let**  $n = \text{lgc-tree2html} ( \langle \langle r, 0 \rangle \rangle , s )$  **in**

**let**  $d = s[\text{'cluster'}][r][r][\text{'diagnose'}]^U$  **in**

**let**  $D = \text{lgc-tree2vector*} ( d , s )$  **in**

**let**  $s = \text{lgc-render-tdiagnose} ( p , D , s )$  **in**

**let**  $s = \text{lgc-render-hdiagnose} ( p , n , d , D , s )$  **in**

**let**  $s = \text{lgc-render-pdiagnose} ( D , s )$  **in**

**let**  $s = \text{lgc-progress} ( \text{'Dumping to cache'} , 3 , s )$  **in**

$\text{lgc-exec-events} ( s , \text{lgc-render-dump} ( x , s ) )]$

Render diagnose. Then pass control to cache dumping.

[lgc-render-tdiagnose (  $p$  ,  $D$  ,  $s$  )  $\equiv$

**let**  $E = \text{fileWrite} ( p :: \text{'diagnose.txt'} , D )$  **in**

$\text{lgc-push-event} ( s , E )]$

Write the diagnose to the rendering directory.

[lgc-render-pdiagnose (  $D$  ,  $s$  )  $\equiv$

**if**  $D$  **then**  $\text{lgc-progress} ( \text{LF} :: \text{'The page is correct'} :: \text{LF} , 2 , s )$  **else**

**let**  $s = \text{lgc-progress} ( \text{LF} :: \text{'Claim failed'} :: \text{LF} , 2 , s )$  **in**

$\text{lgc-progress} ( D , 3 , s )]$

Write the diagnose to standard output (“p” for “progress” or “print”).

## 8.17 HTML rendering of diagnose

```
[lgc-render-hdiagnose ( p , n , d , D , s ) ≐  
  let t = ‘Logiweb diagnose of ’::n in  
  let a = lgc-html-href ( ‘index.html’ , ‘Up’ ) in  
  let a = a::CRLF::lgc-html-help in  
  let d = if d then lgc-render-correct else lgc-render-hdiagnose1 ( D  
  ) in  
  let a = lgc-html-page ( t , a::CRLF::d , s ) in  
  let E = fileWrite ( p::‘diagnose.html’ , a ) in  
  lgc-push-event ( s , E )]
```

Write an html version of the diagnose to the rendering directory.

```
[lgc-render-correct ≐  
  lgc-html-h3 ( ‘The page is correct’ )]  
  Diagnose for correct pages.
```

```
[lgc-render-hdiagnose1 ( D ) ≐  
  lgc-html-p ( lgc-vector*2html ( vt2vector* ( D ) ) )]  
  Diagnose for incorrect pages.
```

## 8.18 Dumping to cache

```
[lgc-render-dump ( x , s ) ≐  
  let r = s[‘reference’] in  
  let p = lgc-render-dirname ( r , s )::rack.lgr in  
  let R = s[‘cluster’][r][r] in  
  let R = lgr-rack-clean ( R ) in  
  let R = rack2sl ( R ) in  
  let s = lgc-progress ( ‘Dumping to ’::p , 4 , s ) in  
  let s = lgc-push-event ( s , fileMkdir ( p ) ) in  
  let s = lgc-push-event ( s , fileWrite ( p , R ) ) in  
  let s = lgc-progress ( ‘User rendering’ , 3 , s ) in  
  lgc-exec-events ( s , lgc-render-user ( x , s ) )]  
  Dump the rack R and pass control to user rendering at end of rack  
  dumping.
```

## 9 User rendering

### 9.1 Overview

Rendering of a page comprises fixed rendering in the rendering directory as defined in Section 8 plus user (i.e. author) defined rendering in a subdirectory named ‘page’ of the rendering directory. By convention, the page directory is supposed to contain a file named ‘index.html’ which is supposed to give a user (i.e. reader) friendly overview of the contents of the page directory. Furthermore, by convention, the page directory is supposed to contain a ‘bin’ directory which is supposed to contain binaries defined on the page.

The author of a page may define his or her own rendering function or may rely on the default provided by the lgc-compiler. In any case, rendering is done in two stages: First, the page is converted into a vector tree  $R$  which contains *rendering events*. Second, the tree  $R$  is converted into output events which are then executed by the Logiweb machine, ultimately resulting in files in the page directory.

Conversion of the tree  $R$  of rendering events to a list of output events is done by `lgc-render-user1 ( R , T , s )` as defined in Section 9.3. This conversion allows the user to produce text files and binary files and to invoke latex, bibtex, makeindex, and dvi<sub>pdf</sub>m.

The difference between text and binary files is in the handling of newlines. When producing a binary file, bytes are written directly to the file. When producing a text file, newline sequences are converted to host newline sequences.

From the point of view of the implementer of Logiweb, the ability to call latex, bibtex, makeindex, and dvi<sub>pdf</sub>m provides a cheap but not completely satisfactory way to produce high quality documents. A more satisfactory solution would be to port those four programs to Logiweb and call them from the renderer inside the system. In that way Logiweb would not rely on external programs and could guarantee that pages would look the same regardless of e.g. which sty files are available at each site.

It should be noted that, once upon a time, Logiweb also allowed to call Mizar and also had facilities for generating MathML, XML, and several other formats. Today, Logiweb just allows the user to define his or her own renderer, so it is up to each author which formats he or she wants to support.

As mentioned, the author of a page may define his or her own rendering function or may rely on the default provided by the lgc-compiler. The conversion is done by `lgc-render-expand ( r , s )` defined in Section 9.6. In case the author has defined a renderer, it is invoked by `lgc-render-expand1 ( d , V , c )`, and otherwise default renderer `lgc-render-default ( d , V , c )` is invoked.

The default renderer renders the body of the page in the page directory and the executables defined on the page in the bin directory under the page directory. The function for rendering of executables, `lgc-render-exec ( r , V , c )` is quite simple. The function for rendering the body, `lgc-render-body ( r , V , c )` is more complicated and resembles the function used for macro expansion on the base page.

However, to speed up rendering, the default renderer is split in two parts: a compiler and an evaluator. The compiler translates rendering definitions (use, show, and name definitions) to *rendering code*. The evaluator evaluates the rendering code. In contrast, the macro expansion engine is an evaluator which directly interprets macro definitions. The reason for splitting rendering into compilation and evaluation is that compilation takes quite some time but only needs to be done once for each construct. Hence, each construct used is compiled and the resulting rendering code is memorized.

The `lgc-render-body ( r , V , c )` function uses `stateexpand ( t , s , c )` to render the body where  $t$  is the body expressed as a term,  $s$  is a *rendering state* and  $c$  is the cache of the page.

A rendering state  $s$  has form  $f :: C :: V$  where  $f$  is a tagged function which is used by default for rendering subexpressions,  $C$  is used for memorizing rendering code, and  $V$  is a value passed down from calling to called renderers. In particular,  $V$ ['parameters'] is supposed to contain the parameters used when invoking `lgc`. This should be used sparingly but may be used, e.g. when there is a need to know the newline convention of the underlying host system or when there is a need to know the location of leap seconds. In addition,  $V$ ['cache'] contains the cache of the page.

## 9.2 Variable names

Default rendering converts a tree  $t$  into a vector tree  $R$  which we refer to as the rendering of  $t$ .

In the default rendering functions, we use parameter names as follows:

$t$	term to be rendered
$R$	rendering
$T$	right hand side of use or show definition
$p = T :: p$	parameter list
$a = T :: a$	argument list
$b = ( T :: t ) :: b$	association list from parameters to arguments
$f$	tagged function
$v$	arbitrary value
$s = f :: v$	rendering state
$c$	cache

## 9.3 Main user rendering functions

```
[lgc-render-user ( x , s ) ⦿
  let r = s['reference'] in
  let E :: R = lgc-render-expand ( r , s )o in
  if E then
    lgc-simple-error ( Exceptionraisedduringuserrendering, goodbye. , s
  ) else
    lgc-render-user1 ( R , T , s )]
```

Perform user rendering, then pass control to `lgc-render-user1` (  $R$  ,  $S$  ,  $s$  ) to get the rendering executed.

```
[lgc-render-user0 (  $R$  )  $\bullet\bullet$ 
  if  $R \in \mathbf{A}$  then  $\top :: \top$  else
  if  $R^h \in \mathbf{Z}$  then  $R$  else lgc-render-user0 (  $R^h$  )]
```

Move down in  $R$  until right above an integer.

```
[lgc-render-noevent  $\bullet\bullet$   $\top$ [
  'file' $\rightarrow$ F][
  'exec' $\rightarrow$ F][
  'text' $\rightarrow$ F][
  'script' $\rightarrow$ F][
  'lgwam' $\rightarrow$ F][
  'latex' $\rightarrow$ F][
  'bibtex' $\rightarrow$ F][
  'makeindex' $\rightarrow$ F][
  'dvipdfm' $\rightarrow$ F]]
```

We have `lgc-render-noevent[e] = F` iff  $e$  names a rendering event.

```
[lgc-render-user1 (  $R$  ,  $S$  ,  $s$  )  $\bullet\bullet$ 
  if  $R \in \mathbf{A}$  then
  if  $S$  then lgc-goodbye (  $s$  ) else lgc-render-user1 (  $S^h$  ,  $S^t$  ,  $s$  ) else
  let  $e :: R = R$  in
  if not  $e \in \mathbf{Z}$  or lgc-render-noevent[e] then lgc-render-user1 (  $e$  ,
   $R :: S$  ,  $s$  ) else let  $s = s$ ['renderstack' $\rightarrow$  $S$ ] in
  let  $r = s$ ['reference'] in
  let  $p = \text{lgc-render-dirname} ( r , s ) :: \text{'page/'}$  in
  let  $x :: R = \text{lgc-render-user0} ( R )$  in
  if  $e = \text{'file'}$  then lgc-render-file (  $p$  ,  $x$  ,  $R$  ,  $s$  ) else
  if  $e = \text{'exec'}$  then lgc-render-file-exec (  $p$  ,  $x$  ,  $R$  ,  $s$  ) else
  if  $e = \text{'text'}$  then lgc-render-text (  $p$  ,  $x$  ,  $R$  ,  $s$  ) else
  if  $e = \text{'script'}$  then lgc-render-text-exec (  $p$  ,  $x$  ,  $R$  ,  $s$  ) else
  if  $e = \text{'lgwam'}$  then lgc-render-lgwam (  $p$  ,  $x$  ,  $R$  ,  $s$  ) else
  if  $e = \text{'latex'}$  then lgc-render-invoke (  $e$  ,  $p$  ,  $x$  ,  $s$  ) else
  if  $e = \text{'bibtex'}$  then lgc-render-invoke (  $e$  ,  $p$  ,  $x$  ,  $s$  ) else
  if  $e = \text{'makeindex'}$  then lgc-render-invoke (  $e$  ,  $p$  ,  $x$  ,  $s$  ) else
  if  $e = \text{'dvipdfm'}$  then lgc-render-invoke (  $e$  ,  $p$  ,  $x$  ,  $s$  ) else let
   $s = \text{lgc-progress} ( \text{'Unknown rendering event:'} :: e , 2 , s )$  in
  lgc-goodbye (  $s$  )]
```

Translate the event  $R$  to an output event followed by passing control to `lgc-render-user2` (  $x$  ,  $s$  ).

```
[lgc-render-user2 (  $x$  ,  $s$  )  $\bullet\bullet$ 
  let  $s = \text{lgc-render-response} ( x , s )$  in
  let  $S = s$ ['renderstack'] in
```

**if**  $S$  **then** lgc-goodbye (  $s$  ) **else**  
 lgc-render-user1 (  $S^h$  ,  $S^t$  ,  $s$  )]

Receive the response from the previous rendering event, then search for the next rendering event.

[lgc-render-response (  $x$  ,  $s$  )  $\doteq$   
**let**  $\langle T, \langle T, r \rangle \rangle = x$  **in**  
**if**  $r = \langle \rangle$  **or**  $r = \langle \text{NULL} \rangle$  **then**  $s$  **else**  
**let**  $\langle p, e, n \rangle = s[\text{'invoked'}]$  **in**  
 lgc-progress ( 'Error running ' : :  $e$  : : SP : :  $n$  : : ' in ' : :  $p$  , 2 ,  $s$  )]  
 Complain if the last rendering event returned a non-zero exit code.

[lgc-goodbye (  $s$  )  $\doteq$   
**if**  $s[\text{'stack'}] \neq T$  **then** lgc-load-fetch (  $s$  ) **else**  
**let**  $s = \text{lgc-progress}$  ( 'Goodbye' , 3 ,  $s$  ) **in**  
 lgc-do-events (  $s$  )]  
 Terminate execution.

[lgc-render-refuse (  $n$  ,  $s$  )  $\doteq$   
**let**  $s = \text{lgc-progress}$  ( 'Improper filename: ' : :  $n$  , 2 ,  $s$  ) **in**  
**let**  $s = \text{lgc-progress}$  ( lgc-render-path (  $n$  ) , 2 ,  $s$  ) **in**  
 lgc-goodbye (  $s$  )]  
 Complain about filename, then quit.

## 9.4 Functions for executing individual rendering events

[lgc-render-file (  $p$  ,  $n$  ,  $c$  ,  $s$  )  $\doteq$   
**if** lgc-render-path (  $n$  )  $\neq T$  **then** lgc-render-refuse (  $n$  ,  $s$  ) **else**  
**let**  $p = p$  : :  $n$  **in**  
**let**  $s = \text{lgc-progress}$  ( 'Writing file: ' : :  $p$  , 4 ,  $s$  ) **in**  
**let**  $E = \text{fileMkdir}$  (  $p$  ) **in**  
**let**  $s = \text{lgc-push-event}$  (  $s$  ,  $E$  ) **in**  
**let**  $E = \text{fileWrite}$  (  $p$  ,  $c$  ) **in**  
**let**  $s = \text{lgc-push-event}$  (  $s$  ,  $E$  ) **in**  
 lgc-exec-events (  $s$  , lgc-render-user2 (  $x$  ,  $s$  ) )]  
 Write contents  $c$  to file named  $n$  relative to path  $p$ .

[lgc-render-file-exec (  $p$  ,  $n$  ,  $c$  ,  $s$  )  $\doteq$   
**if** lgc-render-path (  $n$  )  $\neq T$  **then** lgc-render-refuse (  $n$  ,  $s$  ) **else**  
**let**  $p = p$  : :  $n$  **in**  
**let**  $s = \text{lgc-progress}$  ( 'Writing file: ' : :  $p$  , 4 ,  $s$  ) **in**  
**let**  $E = \text{fileMkdir}$  (  $p$  ) **in**  
**let**  $s = \text{lgc-push-event}$  (  $s$  ,  $E$  ) **in**  
**let**  $E = \text{fileWriteExec}$  (  $p$  ,  $c$  ) **in**  
**let**  $s = \text{lgc-push-event}$  (  $s$  ,  $E$  ) **in**  
 lgc-exec-events (  $s$  , lgc-render-user2 (  $x$  ,  $s$  ) )]

Write contents  $c$  to executable file named  $n$  relative to path  $p$ .

```
[lgc-render-text ( p , n , c , s ) ≐  
  if lgc-render-path ( n ) ≠ T then lgc-render-refuse ( n , s ) else  
  let p = p :: n in  
  let N = lgc-host-newline ( s ) in  
  let s = lgc-progress ( 'Writing file' :: p , 4 , s ) in  
  let E = fileMkdir ( p ) in  
  let s = lgc-push-event ( s , E ) in  
  let E = textWrite ( p , N , c ) in  
  let s = lgc-push-event ( s , E ) in  
  lgc-exec-events ( s , lgc-render-user2 ( x , s ) )]
```

Write contents  $c$  with newline translation to file named  $n$  relative to path  $p$ .

```
[lgc-render-text-exec ( p , n , c , s ) ≐  
  if lgc-render-path ( n ) ≠ T then lgc-render-refuse ( n , s ) else  
  let p = p :: n in  
  let N = lgc-host-newline ( s ) in  
  let s = lgc-progress ( 'Writing file' :: p , 4 , s ) in  
  let E = fileMkdir ( p ) in  
  let s = lgc-push-event ( s , E ) in  
  let E = textWriteExec ( p , N , c ) in  
  let s = lgc-push-event ( s , E ) in  
  lgc-exec-events ( s , lgc-render-user2 ( x , s ) )]
```

Write contents  $c$  with newline translation to executable file named  $n$  relative to path  $p$ .

```
[lgc-render-add-lf ( a ) ≐  
  if a ∈ A then T else  
  ah :: LF :: lgc-render-add-lf ( at )]
```

Add line feed after each element of  $a$ .

```
[lgc-render-lgwam ( p , n , c , s ) ≐  
  if lgc-render-path ( n ) ≠ T then lgc-render-refuse ( n , s ) else  
  let p = p :: n in  
  let N = lgc-host-newline ( s ) in  
  let a = lgc-render-add-lf ( s[‘parameters’][‘script’] ) in  
  let s = lgc-progress ( 'Writing file' :: p , 4 , s ) in  
  let E = fileMkdir ( p ) in  
  let s = lgc-push-event ( s , E ) in  
  let E = textWriteExec ( p , N , a :: c ) in  
  let s = lgc-push-event ( s , E ) in  
  lgc-exec-events ( s , lgc-render-user2 ( x , s ) )]
```

Add script headline  $a$  to contents  $c$ . Then write contents  $c$  with newline translation to executable file named  $n$  relative to path  $p$ . This

is supposed to work under Unix but, in the future, lgc-render-lgwam (  $p$  ,  $n$  ,  $c$  ,  $s$  ) is supposed to be enhanced to work under other operating systems as well. The intension is that an lgwam rendering event should work regardless of operating system.

```
[lgc-render-invoke (  $e$  ,  $p$  ,  $n$  ,  $s$  ) ≡
  if lgc-render-path (  $n$  ) ≠ T then lgc-render-refuse (  $n$  ,  $s$  ) else
  let  $s = s$ [‘invoked’]→⟨ $p$ ,  $e$ ,  $n$ ⟩ in
  let  $s =$  lgc-progress ( ‘Invoking ’ ::  $e$  :: SP ::  $n$  , 4 ,  $s$  ) in
  let  $E =$  execlp1 (  $p$  ,  $e$  ,  $n$  ) in
  let  $s =$  lgc-push-event (  $s$  ,  $E$  ) in
  lgc-exec-events (  $s$  , lgc-render-user2 (  $x$  ,  $s$  ) )]
```

Write contents  $c$  with newline translation to executable file named  $n$  relative to path  $p$ .

## 9.5 Safety check filename

At present, we put restrictions on file names that occur in rendering events. Among other, we do not allow file names like ‘.././foo’. These restrictions prevent users from corrupting their file structure accidentally, but they do not provide any protection against malicious code.

Protection against malicious code requires one to establish a chroot or schroot jail or to abandon use of latex, bibtex, makeindex, and dvi<sub>pdf</sub>m. The latter approach is best, but requires those four programs to be ported to the Logiweb programming language and included in the present compiler.

```
[lgc-render-path (  $n$  ) ≡
  let  $n =$  vt2vector* (  $n$  ) in
  if  $n$  then ‘Empty filename’ else
  if  $n^h =$  ‘/’ then ‘Relative filename starts with slash’ else
  if  $n =$  ⟨‘.’⟩ then ‘Filename must be more than a dot’ else
  lgc-render-path-slash (  $n$  )]
```

Return T if  $n$  is a proper path. Else return error message.

```
[lgc-render-path-slash (  $n$  ) ≡
  if  $n ∈$  A then Filenameendswithaslash else
  if  $n =$  ⟨‘.’⟩ then Filenameendswithslash – dot else
  if  $n^h =$  ‘/’ then Filenamecontainsstwoslashesinarow else
  lgc-render-path1 (  $n$  )]
```

Return T if a slash followed by  $n$  is a proper path. Else return error message.

```
[lgc-render-path1 (  $n$  ) ≡
  if  $n ∈$  A then T else
  let  $c :: n = n$  in
  if ‘a’ ≤  $c$  and  $c ≤$  ‘z’ then lgc-render-path1 (  $n$  ) else
```

```

if 'A' ≤ c and c ≤ 'Z' then lgc-render-path1 ( n ) else
if '0' ≤ c and c ≤ '9' then lgc-render-path1 ( n ) else
if c = '-' or c = '_' then lgc-render-path1 ( n ) else
if c = '.' then lgc-render-path-dot ( n ) else
if c = '/' then lgc-render-path-slash ( n ) else
'Filename must be constructed from letters (a to z and A to Z)'::
LF:::'digits (0 to 9), hyphen (-), underscore (_), dot (.), and slash (/)'

```

Return T if non-slash followed by *n* is a proper path. Else return error message.

```

[lgc-render-path-dot ( n ) ••
if nh = '.' then 'Filename contains two dots in a row' else
lgc-render-path1 ( n )]

```

Return T if a dot followed by *n* is a proper path. Else return error message.

## 9.6 Invokation of user rendering

The lgc-render-expand ( *r* , *s* ) function renders the page with reference *r* for state *s* and returns a list of rendering output events.

```

[lgc-render-expand ( r , s ) ••
let c = s['cluster'] [r] in
let V = T['cache'→c] in
let V = V['parameters'→s['parameters']] in
let V = V['leap'→s['leap']] in
let d = c [r]['codex'] [r] [0] [0] ['render'] in
if d ∈ P then lgc-render-expand1 ( d , V , c ) else
let b = c [r]['bibliography']1 in
if b then lgc-render-default ( r , V , c ) else
let d = c [b]['codex'] [b] [0] [0] ['render'] in
if d ∈ P then lgc-render-expand1 ( d , V , c ) else
lgc-render-default ( r , V , c )]

```

Macro expand page *r* in cache *c*. First look up the macro expander of the page itself and call it *d*. If found, apply it. Else look up the macro expander of the bed page *b* and call it *d*. If found, apply it. Else, default to the identity macro expander and return the body of the page.

```

[lgc-render-expand1 ( d , V , c ) ••
let f = eval ( d3 , T , c ) in
(f " VM)U]

```

Evaluate right hand side of the rendering definition *d*. Then apply the result to the cache *c*.

```
[lgc-render-default ( r , V , c ) ≡
  let R = lgc-render-exec ( r , V , c ) in
  let R = R::lgc-render-lgwinclude ( r , V , c ) in
  let R = R::lgc-render-diagnose ( r , V , c ) in
  R::lgc-render-body ( r , V , c )]
```

Default rendering of a cache  $c$  involves rendering of the body and rendering of executables.

```
[lgc-render-body ( r , V , c ) ≡
  let t = c[r][body] in
  stateexpand ( t , lgc-render-use::T::V , c )h]
```

Default rendering of a body is done by passing the body expressed as a term  $t$  and a rendering state  $f::C::V$  to the `stateexpand` function.

The real work is done by the function  $f$  which defines how to render the root of the body.  $f$  typically renders subterms of the root and then combine renderings of subterms into a rendering of the whole term.

The rendering cache  $C$  of accumulated information is set to `T`. During rendering, information is accumulated in  $C$ . The call to `stateexpand` results in a pair  $R::C$  where  $R$  is the rendering and  $C$  is the final value of  $C$ . The `lgc-render-body ( r , V , c )` function discards this final  $C$  and returns the rendering  $R$ .

The rendering state value  $V$  contains information that may be useful for the renderer. The rendering state value is passed down from calling to called renderer, so  $V$  can only be used for passing information from the root towards the leafs while rendering the body  $t$ .

```
[lgc-render-diagnose ( r , V , c ) ≡
  let t = c[r][diagnose]U in
  if t then lgc-render-diagnose1 ( 'No errors found' , r , V , c ) else
  let R = stateexpand ( t , lgc-render-use::T::V , c )h in
  if tt or not ttt then R else
  let d = c[tr]['codex'][tr][ti][0]['use'] in
  if d or not d21 ≐ d3 then R else
  lgc-render-diagnose1 ( R , r , V , c )]
```

```
[lgc-render-diagnose1 ( m , r , V , c ) ≡
  let N = lgc-render-pdftitle ( vt2vector* ( lgc-symbol2vt1 ( r , 0 , c ) ) ) in
  let n = stateexpand ( <<r,0>> , lgc-render-show::T::V , c )h in
  let t = lgc-lgt2grdutc2vt ( lgc-ref2lgt ( r ) , V ) in
  let R =
  '\documentclass[fleqn]{article}
```

```

\setlength{\overfullrule}{1mm}
\usepackage{latexsym}
\setlength{\parindent}{0em}
\setlength{\parskip}{1ex}
\usepackage{graphicx}
\usepackage[dvipdfm=true]{hyperref}
\hypersetup{pdfpagemode=UseNone}
\hypersetup{pdfstartpage=1}
\hypersetup{pdfstartview=FitBH}
\hypersetup{pdfpagescrop={120 80 490 730}}
\hypersetup{pdftitle=Logiweb diagnose of ' : : N : : ' }
\hypersetup{colorlinks=true}
\everymath{\rm}
\begin{document}\raggedright
\newlength{\lgwlinewidth}
\setlength{\lgwlinewidth}{\linewidth}
\begin{list}{}{\setlength{\leftmargin}{0mm}
\setlength{\rightmargin}{20mm}
\setlength{\topsep}{0mm}
\setlength{\partopsep}{0mm}
\setlength{\itemsep}{0mm}
\setlength{\parsep}{0mm}
}\item \makebox[0mm][l]{\makebox[\lgwlinewidth][r]{\includegraphics
[0mm, 18.5mm][16.5mm, 19mm]{logiweb.eps}}}%
{\bf{\Large Logiweb diagnose of $ ' : : n : : ' $}}
\end{list}\vspace{12.5mm}

' : : m : : '

\vspace{2ex}

{\em\href{http://logiweb.eu/index.html}{The Logiweb compiler (lgc)}
\href{http://logiweb.eu/logiweb/doc/misc/time.html}{' : : t : : '}}
\end{document}

```

```

' in
let R = 'text' : : 'diagnose.tex' : : R in
let R = ⟨R, 'latex', 'diagnose'⟩ in
⟨R, 'dvipdfm', 'diagnose'⟩]

```

```

[lgc-render-pdftitle ( a ) ≡
if a ∈ A then T else
let c : : a = a in
let a = lgc-render-pdftitle ( a ) in
if lgc-render-pdftitle1 ( c ) then c : : a else
(' : : lgc-string2mixed ( c ) : : ' ) : : a]

```

Convert the singleton list  $a$  into a vt which is safe to use as a pdftitle.

```
[lgc-render-pdftitle1 ( c ) ≐
  'a' ≤ c and c ≤ 'z' or
  'A' ≤ c and c ≤ 'Z' or
  '0' ≤ c and c ≤ '9' or
  c = ' ' or c = '.' or c = '-']
```

Return  $\top$  if  $c$  is a singleton which is safe to include in a pdftitle. This function is over-conservative.

## 9.7 Default rendering of executables

```
[lgc-render-exec ( r , V , c ) ≐
  lgc-render-exec1 ( r , V , c[r]['codex'][0] ,  $\top$  )]
```

Render all executables defined by page  $r$ .

```
[lgc-render-exec1 ( r , V , c , R ) ≐
  if c ∈  $\mathbf{A}$  then R else
  if  $c^h \in \mathbf{Z}$  then lgc-render-exec2 ( r , V ,  $c^h$  ,  $c^t$  , R ) else
  lgc-render-exec1 ( r , V ,  $c^h$  , lgc-render-exec1 ( r , V ,  $c^t$  , R ) )]
```

Render all executables defined by page  $r$  in subcodex  $c$  and accumulate the result in  $R$ .

```
[lgc-render-exec2 ( r , V , i , c , R ) ≐
  let t = c[0]['execute']32 in
  if t then R else
  let v = eval ( t ,  $\top$  , V['cache'] )U in
  let v = lgc-render-exec-arg ( v , V ) in
  let a = 'string'::LF::lgc-string2mixed ( r ) in
  let a = a::LF::i::LF::'execute'::v in
  let p = vt2vector ( bin/::i ) in
  ⟨'lgwam', p, a⟩::R]
```

Render executable if defined.

```
[lgc-render-exec-arg ( v , V ) ≐
  if v ∈  $\mathbf{A}$  then LF else
  let a::v = v in
  let w = lgc-render-exec-arg ( v , V ) in
  if a ∈  $\mathbf{Z}$  then LF::a::w else
  if not a ∈  $\mathbf{M}$  then w else
  let a = (a" VM)U in
  if a ∈  $\mathbf{Z}$  then LF::a::w else w]
```

Insert newlines in front of each string in the list  $v$  of strings. Apply maptagged elements to  $V$  first.

## 9.8 Rendering of lgwinclude file

Default rendering includes generation of a file named `lgwinclude.tex`. That file contains  $\text{\TeX}$  macros which can be useful to include. Using `lgwinclude.tex` is not a must, however. One can just as well use advanced rendering functions e.g. for setting `\today` to the timestamp of the page.

```
[lgc-gdef ( x , y )  $\equiv$   $\langle$ '\gdef\ $\backslash$ , x, '{', y, '}'::LF $\rangle$ ]
```

Construct a  $\text{\TeX}$  global definition.

```
[lgc-only-letters ( x )  $\equiv$ 
```

```
  if  $x \in \mathbf{P}$  then lgc-only-letters (  $x^h$  )::lgc-only-letters (  $x^t$  ) else
  if not  $x \in \mathbf{Z}$  then  $\top$  else
  if 'A'  $\leq x$  and  $x \leq$  'Z' or 'a'  $\leq x$  and  $x \leq$  'z' then  $x$  else  $\top$  ]
```

Remove all characters which are not Latin letters from the vector tree  $x$ .

```
[lgc-gdef-ref ( r , c )  $\equiv$ 
```

```
  let  $n = \text{lgc-symbol2vt1} ( r , 0 , c )$  in
  let  $n = \text{vt2vector*} ( n )$  in
  let  $n = \text{lgc-only-letters} ( n )$  in
  let  $X = \text{lgc-string2mixed} ( r )$  in
  let  $R = \text{lgc-gdef} ( \text{'lgwBlock'}::n , X )$  in
   $R::\text{lgc-gdef} ( \text{'lgwBreak'}::n , \text{lgc-break} ( X ) )$ ]
```

Convert the reference  $r$  (given as a vector) into two definitions useful for referencing the page with reference  $r$ .  $c$  must be the cache of the referencing page.

```
[lgc-gdef-bib ( b , c )  $\equiv$ 
```

```
  if  $b \in \mathbf{A}$  then  $\top$  else
  lgc-gdef-ref (  $b^h$  ,  $c$  )::lgc-gdef-bib (  $b^t$  ,  $c$  )]
```

Convert the bibliography  $b$  into two  $\text{\TeX}$  definitions for each reference.  $c$  must be the cache of the referencing page.

```
[lgc-break ( v )  $\equiv$  lgc-break1 ( vt2vector* ( v ) )]
```

```
[lgc-break1 ( v )  $\equiv$ 
```

```
  if  $v^t \in \mathbf{A}$  then  $v$  else
   $v^h::\text{'\lgwbreak'}::\text{LF}::\text{lgc-break1} ( v^t )$ ]
```

```
[lgc-render-lgwinclude ( r , V , c )  $\equiv$ 
```

```
  let  $t = \text{lgc-ref2lgt} ( r )$  in
  let  $R = \text{lgc-gdef} ( \text{'today'} , \text{lgc-lgt2grdutc2vt} ( t , V ) )$  in
  let  $R = R::\text{lgc-gdef} ( \text{'lgwlgt'} , \text{lgc-lgt2vt} ( t ) )$  in
  let  $R = R::\text{lgc-gdef} ( \text{'lgwmjdtai'} , \text{lgc-lgt2mjdtai2vt} ( t ) )$  in
  let  $R = R::\text{lgc-gdef} ( \text{'lgwgrdutc'} , \text{lgc-lgt2grdutc2vt} ( t , V ) )$  in
  let  $\langle f, e \rangle = t$  in
```

```

let R = R::lgc-gdef ( 'lgwmantissa' , lgc-itoa ( f ) ) in
let R = R::lgc-gdef ( 'lgwexponent' , lgc-itoa ( e ) ) in
let ⟨ Y , M , D , h , m , s , f , e ⟩ = lgc-lgt2grdutc ( t , V ) in
let R = R::lgc-gdef ( 'lgwfraction' , lgc-ctoa ( f , e ) ) in
let R = R::lgc-gdef ( 'lgwsecond' , lgc-ctoa ( s , 2 ) ) in
let R = R::lgc-gdef ( 'lgwminute' , lgc-ctoa ( m , 2 ) ) in
let R = R::lgc-gdef ( 'lgwhour' , lgc-ctoa ( h , 2 ) ) in
let R = R::lgc-gdef ( 'lgwday' , lgc-ctoa ( D , 2 ) ) in
let R = R::lgc-gdef ( 'lgwmonth' , lgc-ctoa ( M , 2 ) ) in
let R = R::lgc-gdef ( 'lgwyyear' , lgc-itoa ( Y ) ) in
let R = R::lgc-gdef ( 'lgwbreak' , '\linebreak[0]\hskip0em plus0.5em{
} ) in
let R = R::lgc-gdef ( 'lgwhyphen' , '-' ) in
let R = R::lgc-gdef ( 'lgwunderscore' , '_' ) in
let X = default ( './../logiweb/' , V[ 'parameters' ][ 'relay' ] ) in
let R = R::lgc-gdef ( 'lgwBlockRelay' , X ) in
let R = R::lgc-gdef ( 'lgwBreakRelay' , lgc-break ( X ) ) in
let X = lgc-string2mixed ( r ) in
let R = R::lgc-gdef ( 'lgwBlockThis' , X ) in
let R = R::lgc-gdef ( 'lgwBreakThis' , lgc-break ( X ) ) in
let R = R::lgc-gdef-bib ( c[r][ 'bibliography' ] , c ) in
text::lgwinclude.tex::R]

```

## 9.9 Rendering compiler

The rendering compiler translates a right hand side  $T$  and a parameter list  $p$  of a rendering definition into rendering code.

The constituents of  $T$  are represented thus: A string  $\langle\langle 0, i \rangle\rangle$  is represented by itself. A parameter is represented by the index of the parameter in the parameter list  $p$ . A construct which has no value definition is represented as  $\mathbb{T}$  followed by the translations of subterms. A construct which does have a value definition is represented as the code of that value definition followed by the translations of subterms.

In the latter case above, the code is assumed to be a function which does not depend on its parameters. For that reason, the code is applied to  $\mathbb{T}$  as many times as its arity indicates to get rid of the parameters. Once that is done, the code is assumed to be a map-tagged function of one parameter.

The parameter is supposed to be of form  $a :: s :: c :: T$  where  $s$  in turn is of form  $f :: C :: V$ .  $a$  is the list of subtrees of the construct being rendered.  $\mathbb{T}$  is the list of subtrees of the construct in the rendering definition which is being evaluated.  $c$  is the cache of the page being rendered.  $f$  is a function suggested for rendering of subtrees (i.e. of the elements of  $a$ ).  $C$  is used for memorizing information which only needs to be computed once (like compiled versions of definitions).  $V$  is passed down from root to leaf in the term being rendered.

$[lgc-index ( T , p , r ) \stackrel{\bullet\bullet}{=}$

**if**  $p \in \mathbf{A}$  **then**  $\top$  **else**  
**if**  $T \stackrel{t}{=} p^h$  **then**  $r$  **else**  
lgc-index (  $T$  ,  $p^t$  ,  $r + 1$  )]

Find the position of the term  $T$  in the parameter list  $p$  and return that position. Return  $\top$  if  $T$  is not found.

[lgc-const2val (  $f$  ,  $x$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $x \in \mathbf{A}$  **then**  $f^U$  **else**  
lgc-const2val (  $f$  "  $\top^M$  ,  $x^t$  )]

Convert the code  $f$  of a constant construct to the constant value of the construct. The length of the list  $x$  indicates the arity of the construct.

[lgc-render-compile (  $T$  ,  $p$  ,  $c$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $\langle\langle r, i \rangle\rangle = T$  **in**  
**if**  $r = 0$  **then**  $T$  **else**  
**let**  $f = c[r][\text{'code'}][i]$  **in**  
**if not**  $f \in \mathbf{M}$  **then** lgc-render-compile1 (  $T$  ,  $p$  ,  $c$  ) **else**  
**let**  $f = \text{lgc-const2val} ( f , T^t )$  **in**  
**if not**  $f \in \mathbf{M}$  **then** lgc-render-compile1 (  $T$  ,  $p$  ,  $c$  ) **else**  
 $f :: \text{lgc-render-compile*} ( T^t , p , c )$ ]

Convert the right hand side  $T$  and parameter list  $p$  into a compiled right hand side. In a compiled right hand side, symbols are replaced by their codes, parameters are replaced by their indices, and strings are kept as they are.

[lgc-render-compile1 (  $T$  ,  $p$  ,  $c$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**let**  $t = \text{lgc-index} ( T , p , 0 )$  **in**  
**if**  $t \in \mathbf{Z}$  **then**  $t$  **else**  
 $T :: \text{lgc-render-compile*} ( T^t , p , c )$ ]

Convert the right hand side  $T$  and parameter list  $p$  into a compiled right hand side when  $T$  is known not to have a valid definition.

[lgc-render-compile\* (  $T$  ,  $p$  ,  $c$  )  $\stackrel{\bullet\bullet}{\equiv}$   
**if**  $T \in \mathbf{A}$  **then**  $\top$  **else**  
lgc-render-compile (  $T^h$  ,  $p$  ,  $c$  ) :: lgc-render-compile\* (  $T^t$  ,  $p$  ,  $c$  )  
)]

Coordinatewise application of lgc-render-compile (  $T$  ,  $p$  ,  $c$  ).

## 9.10 Rendering evaluator

The rendering evaluator evaluates rendering code  $T$  for a rendering state  $s$  and cache  $c$ . Furthermore, the evaluator takes a list  $a$  of subtrees of the construct being rendered.

```

[lgc-render-eval ( T , a , s , c )  $\stackrel{\bullet\bullet}{\equiv}$ 
  if  $T \in \mathbf{Z}$  then stateexpand ( nth ( T , a ) , s , c ) else
  let  $f :: T = T$  in
  let  $\top :: C :: V = s$  in
  if  $f \in \mathbf{P}$  then  $f^1 :: C$  else
  if  $f \in \mathbf{M}$  then  $(f \text{''} ((a :: s :: c :: T)^M))^\mathbf{U}$  else
  if  $T$  then  $\top :: C$  else
  if  $T^t$  then lgc-render-eval (  $T^h$  , a , lgc-render-show ::  $C :: V$  , c
  ) else
  lgc-render-eval* ( T , a , s , c )]
```

Apply the compiled right hand side  $T$  to the argument list  $a$ , rendering state  $s$ , and cache  $c$ .

```

[lgc-render-eval* ( T , a , s , c )  $\stackrel{\bullet\bullet}{\equiv}$ 
  let  $f :: C :: V = s$  in
  if  $T \in \mathbf{A}$  then  $\top :: C$  else
  let  $A :: C = \text{lgc-render-eval} ( T^h , a , s , c )$  in
  let  $B :: C = \text{lgc-render-eval*} ( T^t , a , f :: C :: V , c )$  in
  ( $A :: B$ ) ::  $C$ ]
```

Coordinatewise application of  $\text{lgc-render-eval} ( T , a , s , c )$ .

If  $T$  is a string then the function returns the string.

Else, if the root of  $T$  has a value definition then we assume it to be a renderer. A renderer is supposed to be a function whose right hand side has form  $\text{map} ( \lambda u. \dots )$ . Such a function ignores its arguments, so we use  $\text{lgc-const2val} ( f , x )$  to convert the function to its constant value and then apply it to all available information.

Else, we look up  $T$  in the list  $b$  of bindings and render the subtree  $t$  if one is found.

Else, we concatenate the renderings of all subterms of  $T$ .

## 9.11 Use rendering

```

[lgc-render-use  $\stackrel{\bullet\bullet}{\equiv}$  map (  $\lambda x. \text{lgc-render-use1} ( x )$  )]
```

Rendering state for use rendering.

```

[lgc-render-use1 ( x )  $\stackrel{\bullet\bullet}{\equiv}$ 
  let  $\langle t , s , c \rangle = x$  in
  let  $f :: C :: V = s$  in
  if  $t$  then  $\text{\texttt{\char34}} :: C$  else
  if  $t^h$  then lgc-understood ( lgc-render-use1 (  $\langle t^1 , s , c \rangle$  ) ) ::  $C$  else
  let  $\langle \langle r , i \rangle \rangle = t$  in
  if  $r = 0$  then  $i :: C$  else
  let  $T = C[\text{\texttt{\char34}}][r][i]$  in
  if not  $T$  then lgc-render-eval ( T ,  $t^t$  , s , c ) else
```

**let**  $T :: C = \text{lgc-render-compile-use} ( r , i , C , c )$  **in**  
 $\text{lgc-render-eval} ( T , t^t , f :: C :: V , c )]$

Produce the rendering of the tree  $t$  for the rendering state  $s$  and the cache  $c$ . The function has to treat certain malformed trees in certain ways: A value of  $T$  has to be rendered as a double quote because the Logiweb compiler uses  $T$  as a placeholder when rendering partial trees. The Logiweb compiler uses that e.g. when rendering the Logiweb symbols in the dictionary of a page. Furthermore, a tree whose root is  $T$  should be enclosed in blue brackets because the Logiweb compiler represents an understood parenthesis as the root  $T$  with one subtree. The Logiweb compiler uses that e.g. when rendering the diagnose of a faulty page.

$[\text{lgc-render-compile-use} ( r , i , C , c ) \equiv$   
**let**  $T :: C = \text{lgc-render-compile-use1} ( r , i , C , c )$  **in**  
**let**  $C = C[⟨\text{'use'}, r, i⟩ \Rightarrow T]$  **in**  $T :: C]$

Compile the use definition of the construct with reference  $r$  and index  $i$  into rendering code and memorize the result in  $C$ .

$[\text{lgc-render-compile-use1} ( r , i , C , c ) \equiv$   
**let**  $d = c[r][\text{codex}][r][i][0][\text{'use'}]$  **in**  
**if**  $d \in \mathbf{P}$  **then**  $\text{lgc-render-compile} ( d^3 , d^{2t} , c ) :: C$  **else**  
 $\text{lgc-render-compile-show} ( r , i , C , c )]$

Compile the use definition of the construct with reference  $r$  and index  $i$  into rendering code.

$[\text{lgc-understood} ( t ) \equiv$   
 $\text{LF} :: \backslash \text{\linebreak[0]\textcolor\{blue\}\{\}\backslash\linebreak[0]\ ' :: t ::$   
 $\text{LF} :: \backslash \text{\linebreak[0]\textcolor\{blue\}\{\}\backslash\linebreak[0]\ ' ]$

Function for enclosing  $t$  in blue brackets.

## 9.12 Show rendering

$[\text{lgc-render-show} \equiv \text{map} ( \lambda x. \text{lgc-render-show1} ( x ) )]$

Rendering state for show rendering.

$[\text{lgc-render-show1} ( x ) \equiv$   
**let**  $\langle t, s, c \rangle = x$  **in**  
**let**  $f :: C :: V = s$  **in**  
**if**  $t$  **then**  $\backslash \text{\mbox\{\tt\char34\}} :: C$  **else**  
**if**  $t^h$  **then**  $\text{lgc-understood} ( \text{lgc-render-show1} ( \langle t^1, s, c \rangle ) ) :: C$  **else**  
**let**  $\langle \langle r, i \rangle \rangle = t$  **in**  
**if**  $r = 0$  **then**  $\text{lgc-render-string} ( i ) :: C$  **else**  
**let**  $T = C[\text{'show'}][r][i]$  **in**  
**if not**  $T$  **then**  $\text{lgc-render-eval} ( T , t^t , s , c )$  **else**

**let**  $T :: C = \text{lgc-render-compile-show} ( r , i , C , c )$  **in**  
 $\text{lgc-render-eval} ( T , t^t , f :: C :: V , c )]$

Produce the rendering of the tree  $t$  for the rendering state  $s$  and the cache  $c$ . The function has to treat certain malformed trees in certain ways: A value of  $\mathbf{T}$  has to be rendered as a double quote because the Logiweb compiler uses  $\mathbf{T}$  as a placeholder when rendering partial trees. The Logiweb compiler uses that e.g. when rendering the Logiweb symbols in the dictionary of a page. Furthermore, a tree whose root is  $\mathbf{T}$  should be enclosed in blue brackets because the Logiweb compiler represents an understood parenthesis as the root  $\mathbf{T}$  with one subtree. The Logiweb compiler uses that e.g. when rendering the diagnose of a faulty page.

$[\text{lgc-render-compile-show} ( r , i , C , c ) \equiv$   
**let**  $T = \text{lgc-render-compile-show1} ( r , i , c )$  **in**  
**let**  $C = C[('show', r, i) \Rightarrow T]$  **in**  $T :: C]$

$[\text{lgc-render-compile-show1} ( r , i , c ) \equiv$   
**let**  $d = c[r][\text{codex}][r][i][0][('show')]$  **in**  
**if**  $d \in \mathbf{P}$  **then**  $\text{lgc-render-compile} ( d^3 , d^{2t} , c )$  **else**  
 $\text{lgc-render-name} ( r , i , c )]$

Compile the show definition of the construct with reference  $r$  and index  $i$  into rendering code.

### 9.13 Rendering based on name

$[\text{lgc-render-name} ( r , i , c ) \equiv$   
**let**  $n = \text{lgc-symbol2vt1} ( r , i , c )$  **in**  
**let**  $n = \text{vt2vector*} ( n )$  **in**  
**let**  $n = \text{lgc-render-name1} ( n , \mathbf{T} )$  **in**  
 $\mathbf{T} :: \langle \langle 0, n^h \rangle \rangle :: \text{lgc-render-name2} ( n^t , 0 )]$

Function for compiling constructs that do not have a tex aspect.

$[\text{lgc-render-name1} ( n , r ) \equiv$   
**if**  $n \in \mathbf{A}$  **then**  $\langle \text{lgc-render-string0} ( \text{reverse} ( r ) ) \rangle$  **else**  
**let**  $c :: n = n$  **in**  
**if**  $c \neq \text{QQ}$  **then**  $\text{lgc-render-name1} ( n , c :: r )$  **else**  
 $\text{lgc-render-string0} ( \text{reverse} ( r ) ) :: \text{lgc-render-name1} ( n , \mathbf{T} )]$

Convert the name  $n$  into a list of inter-quote strings.

$[\text{lgc-render-name2} ( n , i ) \equiv$   
**if**  $n \in \mathbf{A}$  **then**  $\mathbf{T}$  **else**  
 $i :: \langle \langle 0, n^h \rangle \rangle :: \text{lgc-render-name2} ( n^t , i + 1 )]$

Merge the list  $n$  of inter-quote strings with parameter indexes.

## 9.14 Show rendering of strings

[lgc-render-string ( *i* )  $\equiv$

```

let i = vt2vector* ( i ) in
let i = lgc-render-string0 ( i ) in
‘\mbox{‘}’ :: i :: ‘\mbox{’}’

```

Render the string *i*.

[lgc-render-string0 ( *i* )  $\equiv$

```

lgc-render-string1 ( i , lgc-render-array ,  $\top$  )]

```

[lgc-render-string1 ( *i* , *a* , *b* )  $\equiv$

```

if i ∈ A then lgc-render-string2 ( b ,  $\top$  ) else
let c :: i = i in
let d = a[c] in
if d then lgc-render-string2 ( c :: b , lgc-render-string0 ( i ) ) else
if d ∈ Z then d :: lgc-render-string0 ( i ) else
lgc-render-string1 ( i , d , c :: b )]

```

Look up the byte sequence at the beginning of *i* in the trie *a* and, simultaneously, accumulate the looked up bytes in *b*. If the array provides a translation (i.e. if *a*[*c*] is a string) then include that translation in the result and discard *b*. If the array provides no translation (i.e. if *a*[*c*] =  $\top$ ) then give up and render *b* as a sequence of bytes. Otherwise, look at one more byte.

[lgc-render-string2 ( *b* , *i* )  $\equiv$

```

if b ∈ A then i else
let c :: b = b in
let m :: n = floor ( c - NULL , 16 ) in
let m = lgc-hexdigit ( m ) in
let n = lgc-hexdigit ( n ) in
lgc-render-string2 ( b , ‘(‘ :: m :: n :: ‘)’ :: i )]

```

Render *b* as a sequence of bytes and revappend them to *i*.

[lgc-render-array  $\equiv$

```

 $\top$ 
[LF → vt2vector ( LF :: ‘\newline ’ )]
[‘ ’ → vt2vector ( ‘\linebreak [0]\ ’ )]
[‘!’ → ‘!’]
[‘"’ → ‘\mbox{\tt\char34}’]
[‘#’ → ‘\#’]
[‘$’ → ‘\$’]
[‘%’ → ‘\%’]
[‘&’ → ‘\&’]
[‘”’ → ‘\mbox{’}’]
[‘(’ → ‘(’]

```

[‘)’→‘)’]  
 [‘\*’→‘{\*}’]  
 [‘+’→‘{+}’]  
 [‘,’→‘,’]  
 [‘-’→‘\mbox{-}’]  
 [‘.’→‘.’]  
 [‘/’→‘/’]  
 [‘0’→‘0’]  
 [‘1’→‘1’]  
 [‘2’→‘2’]  
 [‘3’→‘3’]  
 [‘4’→‘4’]  
 [‘5’→‘5’]  
 [‘6’→‘6’]  
 [‘7’→‘7’]  
 [‘8’→‘8’]  
 [‘9’→‘9’]  
 [‘:’→‘{:}’]  
 [‘;’→‘;’]  
 [‘<’→‘<’]  
 [‘=’→‘{=}’]  
 [‘>’→‘>’]  
 [‘?’→‘?’]  
 [‘@’→‘@’]  
 [‘A’→‘A’]  
 [‘B’→‘B’]  
 [‘C’→‘C’]  
 [‘D’→‘D’]  
 [‘E’→‘E’]  
 [‘F’→‘F’]  
 [‘G’→‘G’]  
 [‘H’→‘H’]  
 [‘I’→‘I’]  
 [‘J’→‘J’]  
 [‘K’→‘K’]  
 [‘L’→‘L’]  
 [‘M’→‘M’]  
 [‘N’→‘N’]  
 [‘O’→‘O’]  
 [‘P’→‘P’]  
 [‘Q’→‘Q’]  
 [‘R’→‘R’]  
 [‘S’→‘S’]  
 [‘T’→‘T’]  
 [‘U’→‘U’]  
 [‘V’→‘V’]

['W'→'W']  
 ['X'→'X']  
 ['Y'→'Y']  
 ['Z'→'Z']  
 ['[ '→'[']  
 ['\ '→'\mbox{\$\backslash\$}']  
 [']'→']']  
 ['^'→'\char94']  
 ['\_ '→'\\_']  
 ['{ '→'\mbox{' '}']  
 ['a'→'a']  
 ['b'→'b']  
 ['c'→'c']  
 ['d'→'d']  
 ['e'→'e']  
 ['f'→'f']  
 ['g'→'g']  
 ['h'→'h']  
 ['i'→'i']  
 ['j'→'j']  
 ['k'→'k']  
 ['l'→'l']  
 ['m'→'m']  
 ['n'→'n']  
 ['o'→'o']  
 ['p'→'p']  
 ['q'→'q']  
 ['r'→'r']  
 ['s'→'s']  
 ['t'→'t']  
 ['u'→'u']  
 ['v'→'v']  
 ['w'→'w']  
 ['x'→'x']  
 ['y'→'y']  
 ['z'→'z']  
 ['{'→'\{']  
 ['|'→'|']  
 ['}'→'\}']  
 ['~'→'\char126']  
 [⟨NULL + 195, NULL + 198⟩⇒'\mbox{AE}']  
 [⟨NULL + 195, NULL + 216⟩⇒'\mbox{O}']  
 [⟨NULL + 195, NULL + 197⟩⇒'\mbox{AA}']  
 [⟨NULL + 195, NULL + 230⟩⇒'\mbox{ae}']  
 [⟨NULL + 195, NULL + 248⟩⇒'\mbox{o}']  
 [⟨NULL + 195, NULL + 229⟩⇒'\mbox{aa}']

]

Typesetting of ASCII and Danish characters. The Danish characters are included to show how multibyte characters are treated. Once the present source is expressed in UTF-8 one can replace e.g.  $\langle \text{NULL} + 195, \text{NULL} + 198 \rangle$  with  $\text{vt2vector}^*( 'x' )$  where  $x$  should be replaced by  $\text{Æ}$ .

## 10 Test cases

### 10.1 Lexical Analysis

```
[lgc-test ( f ) ≡
  let f = vt2vector* ( f ) in
  let s = T[‘source’→f] in
  let s = s[‘verbose’→3] in
  let e :: S = lgc-lex-source ( f , s )° in
  let s = if e then s[‘msg’→S] else S in
  s[‘body’→reverse ( s[‘body’] )]]
```

The function above invokes lexical analysis on the source text  $f$ .

At some time in the past,  $s[‘body’]$  was reversed so that it appears in natural order. But the test cases below still compare  $s[‘body’]$  to reversed lists, so the function above reverses  $s[‘body’]$  to make it work with the old test cases below.

$[lgc-test ( ‘abc’ )[‘body’] = \langle \langle ‘c’, 2 \rangle, \langle ‘b’, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, SP, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 2 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, SP, SP, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 3 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle SP, SP, ‘a’, SP, SP, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 5 \rangle, \langle SP, 3 \rangle, \langle ‘a’, 2 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, TAB, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 2 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, FF, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 2 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, CR, LF, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 3 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, LF, CR, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 3 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, CR, LF, CR, LF, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 5 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, CR, LF, LF, CR, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 5 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, LF, CR, CR, LF, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 5 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, LF, CR, LF, CR, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 5 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \langle ‘a’, LF, FF, CR, ‘b’ \rangle )[‘body’] = \langle \langle ‘b’, 4 \rangle, \langle SP, 1 \rangle, \langle ‘a’, 0 \rangle \rangle =$

$[lgc-test ( \text{""C""N""S} )['body']] = \langle \langle lgc-esc-S, 6 \rangle, \langle lgc-esc-N, 3 \rangle, \langle lgc-esc-C, 0 \rangle \rangle =$   
 $[lgc-test ( \langle 'a''; b', LF, 'c' \rangle )['body']] = \langle \langle 'c', 6 \rangle, \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''\{b''\}c' )['body']] = \langle \langle 'c', 8 \rangle, \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''\{b''''\}c''d' )['body']] = \langle \langle 'd', 13 \rangle, \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''bc"d' )['body']] = \langle \langle 'd', 5 \rangle, lgc-esc--::1::'bc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''-bc"d' )['body']] = \langle \langle 'd', 7 \rangle, lgc-esc--::1::'bc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''\$bc"d' )['body']] = \langle \langle 'd', 7 \rangle, lgc-esc-\$::1::'bc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''\#bc"d' )['body']] = \langle \langle 'd', 7 \rangle, lgc-esc-\#::1::'bc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''\#bc''d' )['body']] = \langle \langle 'd', 9 \rangle, lgc-esc-\#::1::'bc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''nc"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::LF::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''!c"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::QQ::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''-c"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''rc"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::CR::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''fc"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::FF::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''tc"d' )['body']] = \langle \langle 'd', 8 \rangle, lgc-esc--::1::vt2vector ( 'b'::TAB::'c' ), \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''x41.c"d' )['body']] = \langle \langle 'd', 11 \rangle, lgc-esc--::1::'bAc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''b''x4142.c"d' )['body']] = \langle \langle 'd', 13 \rangle, lgc-esc--::1::'bABc', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( 'a''-''x4142."d' )['body']] = \langle \langle 'd', 13 \rangle, lgc-esc--::1::'AB', \langle 'a', 0 \rangle \rangle =$   
 $[lgc-test ( \langle QQ, SP, QQ \rangle )['body']] = \langle lgc-esc--::0::SP \rangle =$   
 $[lgc-test ( \langle QQ, SP, SP, QQ \rangle )['body']] = \langle lgc-esc--::0::' \rangle =$   
 $[lgc-test ( \langle QQ, TAB, QQ \rangle )['body']] = \langle lgc-esc--::0::SP \rangle =$   
 $[lgc-test ( \langle QQ, FF, QQ \rangle )['body']] = \langle lgc-esc--::0::vt2vector ( LF ) \rangle =$   
 $[lgc-test ( \langle QQ, CR, LF, QQ \rangle )['body']] = \langle lgc-esc--::0::vt2vector ( LF ) \rangle =$   
 $[lgc-test ( \langle QQ, LF, CR, QQ \rangle )['body']] = \langle lgc-esc--::0::vt2vector ( LF ) \rangle =$

[lgc-test ( (QQ, CR, LF, CR, LF, QQ) )['body'] = ⟨lgc-esc-- :: 0 :: vt2vector ( LF :: LF )⟩=

[lgc-test ( (QQ, CR, LF, LF, CR, QQ) )['body'] = ⟨lgc-esc-- :: 0 :: vt2vector ( LF :: LF )⟩=

[lgc-test ( (QQ, LF, CR, CR, LF, QQ) )['body'] = ⟨lgc-esc-- :: 0 :: vt2vector ( LF :: LF )⟩=

[lgc-test ( (QQ, LF, CR, LF, CR, QQ) )['body'] = ⟨lgc-esc-- :: 0 :: vt2vector ( LF :: LF )⟩=

[lgc-test ( (QQ, LF, FF, CR, QQ) )['body'] = ⟨lgc-esc-- :: 0 :: vt2vector ( LF :: LF :: LF )⟩=

[lgc-test ( 'a""b' )['body'] = ⟨⟨'b', 4⟩, lgc-esc-- :: 1 :: ' ', ⟨'a', 0⟩⟩=

[lgc-test ( 'ab""Pcd  
ef' )['body'] = ⟨⟨'f', 9⟩, ⟨'e', 8⟩, ⟨'b', 1⟩, ⟨'a', 0⟩⟩=

[lgc-test ( 'ab""Pcd  
ef' )['page'] = ⟨⟨'c', 'd'⟩⟩=

[lgc-test ( 'ab""P c d  
ef' )['page'] = ⟨⟨'c', SP, 'd'⟩⟩=

[lgc-test ( 'ab""Pab"cd"ef  
ef' )['page'] = ⟨⟨'e', 'f'⟩, ⟨'c', 'd'⟩, ⟨'a', 'b'⟩⟩=

[lgc-test ( 'ab""P a b " c d " e f  
ef' )['page'] = ⟨⟨'e', SP, 'f'⟩, ⟨SP, 'c', SP, 'd', SP⟩, ⟨SP, 'a', SP, 'b', SP⟩⟩=

[lgc-test ( 'abyz' )['bib'] = ⟨⟩=

[lgc-test ( 'ab""Rcd  
yz' )['bib'] = ⟨⟨⟨'c', 'd'⟩⟩⟩=

[lgc-test ( 'ab""Rcd  
""Ref  
yz' )['bib'] = ⟨⟨⟨'e', 'f'⟩⟩, ⟨⟨'c', 'd'⟩⟩⟩=

[lgc-test ( 'ab""Rcd"ef"gh  
""Rij  
yz' )['bib'] = ⟨⟨⟨'i', 'j'⟩⟩, ⟨⟨'g', 'h'⟩, ⟨'e', 'f'⟩, ⟨'c', 'd'⟩⟩⟩=

[lgc-test ( 'ab""R c d " e f " g h  
""Rij  
yz' )['bib'] = ⟨⟨⟨'i', 'j'⟩⟩, ⟨⟨'g', SP, 'h'⟩, ⟨SP, 'e', SP, 'f', SP⟩, ⟨SP, 'c', SP, 'd'⟩⟩⟩=

```
[lgc-test ( 'ab""D1
  c d " e f " g h
  "ij"kl"
  ""D2
  mn""Byz' )['def'] = <<<(45, 2), <(m, n)>, <(5, 1)>, <c, SP, d, SP, QQ, SP, e, SP
```

```
[lgc-test ( 'ab""D c d " e f " g h
  "ij"kl"
  ""Dmn""Byz' )['body'] = <<<'z', 49>, <'y', 48>, <'b', 1>, <'a', 0>)]="
```

## 10.2 Splicing

```
[lgc-splice ( vt2vector* ( ' base ' ) , vt2vector* ( 'if " then " else "' ) ) =
  vt2vector* ( 'base if " then " else "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base ' ) , vt2vector* ( 'if " then " else "' ) ) =
  vt2vector* ( 'base if " then " else "' )]=
```

```
[lgc-splice ( vt2vector* ( ' base' ) , vt2vector* ( 'if " then " else "' ) ) =
  vt2vector* ( 'baseif " then " else "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base' ) , vt2vector* ( 'if " then " else "' ) ) = vt2vector*
  ( 'baseif " then " else "' )]=
```

```
[lgc-splice ( vt2vector* ( ' base ' ) , vt2vector* ( "" + "' ) ) = vt2vector* ( ""
  base + "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base ' ) , vt2vector* ( "" + "' ) ) = vt2vector* ( ""
  base + "' )]=
```

```
[lgc-splice ( vt2vector* ( ' base' ) , vt2vector* ( "" + "' ) ) = vt2vector* ( ""
  base+ "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base' ) , vt2vector* ( "" + "' ) ) = vt2vector* ( ""
  base+ "' )]=
```

```
[lgc-splice ( vt2vector* ( ' base ' ) , vt2vector* ( "", "' ) ) = vt2vector* ( "" base
  , "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base ' ) , vt2vector* ( "", "' ) ) = vt2vector* ( ""base
  , "' )]=
```

```
[lgc-splice ( vt2vector* ( ' base' ) , vt2vector* ( "", "' ) ) = vt2vector* ( ""
  base, "' )]=
```

```
[lgc-splice ( vt2vector* ( 'base' ) , vt2vector* ( "", "' ) ) = vt2vector* ( ""base, "'
  )]=
```

```
[lgc-splice ( vt2vector* ( ' base ' ) , vt2vector* ( "" "' ) ) = vt2vector* ( "" base
  "' )]=
```

[lgc-splice ( vt2vector\* ( 'base' ) , vt2vector\* ( " " ) ) = vt2vector\* ( " base  
" )]=

[lgc-splice ( vt2vector\* ( ' base' ) , vt2vector\* ( " " ) ) = vt2vector\* ( " base  
" )]=

[lgc-splice ( vt2vector\* ( 'base' ) , vt2vector\* ( " " ) ) = vt2vector\* ( " base  
" )]=

### 10.3 Auxiliary definitions

[selfref  $\stackrel{\bullet\bullet}{\equiv}$  self[0]]

[baseref  $\stackrel{\bullet\bullet}{\equiv}$  base[0]]

[lgw-trisect-self  $\stackrel{\bullet\bullet}{\equiv}$   
let  $v$  = self[selfref][‘vector’] in  
let  $c$  = lgw-trisect ( vt2vector\* (  $v$  ) ) in  
 $c$ [baseref $\rightarrow$ base[baseref]]]

[lgw-trisect-base  $\stackrel{\bullet\bullet}{\equiv}$   
lgw-trisect ( vt2vector\* ( base[baseref][‘vector’] ) )]

[lgw-codify-self  $\stackrel{\bullet\bullet}{\equiv}$  lgw-codify ( selfref , lgw-trisect-self , 4 )]

[lgw-codify-base  $\stackrel{\bullet\bullet}{\equiv}$  lgw-codify ( baseref , lgw-trisect-base , 4 )]

### 10.4 Trisecting of lgw files

[lgw-parse-string ( bt2vector\* (  $\langle 2, 65, 66, 67 \rangle$  ) ) =  $\langle$ ‘AB’, C’ $\rangle$ ]=

[lgw-parse-bibliography ( bt2vector\* (  $\langle 1, 65, 1, 66, 0, 67 \rangle$  ) ) =  $\langle$ ‘A’, ‘B’, ‘C’ $\rangle$ ]=

[lgw-parse-dictionary ( bt2vector\* (  $\langle 1, 0, 2, 1, 0, 67 \rangle$  ) ) =  $\langle$ T[0 $\rightarrow$ 0][1 $\rightarrow$ 0][2 $\rightarrow$ 1], C’ $\rangle$ ]=

[lgw-trisect ( bt2vector\* (  $\langle 1, 65, 1, 66, 0, 1, 0, 2, 1, 0, 67 \rangle$  ) ) = T[0 $\rightarrow$ ‘A’][‘A’, ‘vector’]  
(  $\langle 1, 65, 1, 66, 0, 1, 0, 2, 1, 0, 67 \rangle$  )][‘A’, ‘bibliography’] $\Rightarrow$ ‘A’, ‘B’][‘A’, ‘d  
[‘A’, ‘dictionary’, 1] $\Rightarrow$ 0][‘A’, ‘dictionary’, 2] $\Rightarrow$ 1][‘A’, ‘body’] $\Rightarrow$ ‘C’]]]=

### 10.5 Test of conversion from GRD to MJD

[lgc-grd2mjd (  $\langle 1858, 11, 17 \rangle$  ) = 0]=

[lgc-grd2mjd (  $\langle 1858, 11, 18 \rangle$  ) = 1]=

[lgc-grd2mjd (  $\langle 1858, 11, 20 \rangle$  ) = 3]=

[lgc-grd2mjd (  $\langle 1858, 11, 30 \rangle$  ) = 13]=

[lgc-grd2mjd (  $\langle 1858, 12, 01 \rangle$  ) = 14]=

$$\begin{aligned}
[\text{lgc-grd2mjd}(\langle 1858, 12, 31 \rangle)] &= 44]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 01, 01 \rangle)] &= 45]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 02, 01 \rangle)] &= 45 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 03, 01 \rangle)] &= 45 + 31 + 28]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 04, 01 \rangle)] &= 45 + 31 + 28 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 05, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 06, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 07, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 08, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 09, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 10, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 11, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1859, 12, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 01, 01 \rangle)] &= 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 01, 01 \rangle)] &= 45 + 365]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 02, 01 \rangle)] &= 45 + 365 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 03, 01 \rangle)] &= 45 + 365 + 31 + 29]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 04, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 05, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 06, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 07, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 08, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 09, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 10, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 11, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31]^\text{=} \\
[\text{lgc-grd2mjd}(\langle 1860, 12, 01 \rangle)] &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30]^\text{=}
\end{aligned}$$

$$\begin{aligned}
[\text{lgc-grd2mjd} (\langle 1861, 01, 01 \rangle) &= 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + \\
&31 + 30 + 31 + 30 + 31] = \\
[\text{lgc-grd2mjd} (\langle 1861, 01, 01 \rangle) &= 45 + 365 + 366] = \\
[\text{lgc-grd2mjd} (\langle 1862, 01, 01 \rangle) &= 45 + 365 + 366 + 365] = \\
[\text{lgc-grd2mjd} (\langle 1863, 01, 01 \rangle) &= 45 + 365 + 366 + 365 + 365] = \\
[\text{lgc-grd2mjd} (\langle 1864, 01, 01 \rangle) &= 45 + 365 + 366 + 365 + 365 + 365] = \\
[\text{lgc-grd2mjd} (\langle 1865, 01, 01 \rangle) &= 45 + 365 + 366 + 365 + 365 + 365 + 366] = \\
[\text{lgc-grd2mjd} (\langle 1900, 01, 01 \rangle) &= 15020] = \\
[\text{lgc-grd2mjd} (\langle 1900, 02, 28 \rangle) &= 15078] = \\
[\text{lgc-grd2mjd} (\langle 1900, 03, 01 \rangle) &= 15079] = \\
[\text{lgc-grd2mjd} (\langle 2000, 01, 01 \rangle) &= 51544] = \\
[\text{lgc-grd2mjd} (\langle 2000, 02, 28 \rangle) &= 51602] = \\
[\text{lgc-grd2mjd} (\langle 2000, 03, 01 \rangle) &= 51604] = \\
[\text{lgc-grd2mjd} (\langle 2100, 02, 28 \rangle) &= 88127] = \\
[\text{lgc-grd2mjd} (\langle 2100, 03, 01 \rangle) &= 88128] = \\
[\text{lgc-grd2mjd} (\langle 2200, 02, 28 \rangle) &= 124651] = \\
[\text{lgc-grd2mjd} (\langle 2200, 03, 01 \rangle) &= 124652] = \\
[\text{lgc-grd2mjd} (\langle 2300, 02, 28 \rangle) &= 161175] = \\
[\text{lgc-grd2mjd} (\langle 2300, 03, 01 \rangle) &= 161176] = \\
[\text{lgc-grd2mjd} (\langle 2400, 02, 28 \rangle) &= 197699] = \\
[\text{lgc-grd2mjd} (\langle 2400, 03, 01 \rangle) &= 197701] =
\end{aligned}$$

## 10.6 Test of conversion from MJD to GRD

$$\begin{aligned}
[\langle 1858, 11, 17 \rangle = \text{lgc-mjd2grd} ( 0 )] &= \\
[\langle 1858, 11, 18 \rangle = \text{lgc-mjd2grd} ( 1 )] &= \\
[\langle 1858, 11, 20 \rangle = \text{lgc-mjd2grd} ( 3 )] &= \\
[\langle 1858, 11, 30 \rangle = \text{lgc-mjd2grd} ( 13 )] &= \\
[\langle 1858, 12, 01 \rangle = \text{lgc-mjd2grd} ( 14 )] &= \\
[\langle 1858, 12, 31 \rangle = \text{lgc-mjd2grd} ( 44 )] &=
\end{aligned}$$

$$\begin{aligned}
&[(1859, 01, 01) = \text{lgc-mjd2grd} ( 45 )]= \\
&[(1859, 02, 01) = \text{lgc-mjd2grd} ( 45 + 31 )]= \\
&[(1859, 03, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 )]= \\
&[(1859, 04, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 )]= \\
&[(1859, 05, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 )]= \\
&[(1859, 06, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 )]= \\
&[(1859, 07, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 )]= \\
&[(1859, 08, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 )]= \\
&[(1859, 09, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 )]= \\
&[(1859, 10, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 \\
&\quad )]= \\
&[(1859, 11, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 \\
&\quad )]= \\
&[(1859, 12, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30 \\
&\quad )]= \\
&[(1860, 01, 01) = \text{lgc-mjd2grd} ( 45 + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + \\
&\quad 31 + 30 + 31 )]= \\
&[(1860, 01, 01) = \text{lgc-mjd2grd} ( 45 + 365 )]= \\
&[(1860, 02, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 )]= \\
&[(1860, 03, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 )]= \\
&[(1860, 04, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 )]= \\
&[(1860, 05, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 )]= \\
&[(1860, 06, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 )]= \\
&[(1860, 07, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 )]= \\
&[(1860, 08, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 )]= \\
&[(1860, 09, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 \\
&\quad )]= \\
&[(1860, 10, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30 \\
&\quad )]= \\
&[(1860, 11, 01) = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 \\
&\quad )]=
\end{aligned}$$

$$\begin{aligned}
\llbracket \langle 1860, 12, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + \\
31 + 30 + 31 + 30 ) \rrbracket = \\
\llbracket \langle 1861, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 31 + 29 + 31 + 30 + 31 + 30 + 31 + \\
31 + 30 + 31 + 30 + 31 ) \rrbracket = \\
\llbracket \langle 1861, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 366 ) \rrbracket = \\
\llbracket \langle 1862, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 366 + 365 ) \rrbracket = \\
\llbracket \langle 1863, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 366 + 365 + 365 ) \rrbracket = \\
\llbracket \langle 1864, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 366 + 365 + 365 + 365 ) \rrbracket = \\
\llbracket \langle 1865, 01, 01 \rangle = \text{lgc-mjd2grd} ( 45 + 365 + 366 + 365 + 365 + 365 + 366 ) \rrbracket = \\
\llbracket \langle 1900, 01, 01 \rangle = \text{lgc-mjd2grd} ( 15020 ) \rrbracket = \\
\llbracket \langle 1900, 02, 28 \rangle = \text{lgc-mjd2grd} ( 15078 ) \rrbracket = \\
\llbracket \langle 1900, 03, 01 \rangle = \text{lgc-mjd2grd} ( 15079 ) \rrbracket = \\
\llbracket \langle 2000, 01, 01 \rangle = \text{lgc-mjd2grd} ( 51544 ) \rrbracket = \\
\llbracket \langle 2000, 02, 28 \rangle = \text{lgc-mjd2grd} ( 51602 ) \rrbracket = \\
\llbracket \langle 2000, 03, 01 \rangle = \text{lgc-mjd2grd} ( 51604 ) \rrbracket = \\
\llbracket \langle 2100, 02, 28 \rangle = \text{lgc-mjd2grd} ( 88127 ) \rrbracket = \\
\llbracket \langle 2100, 03, 01 \rangle = \text{lgc-mjd2grd} ( 88128 ) \rrbracket = \\
\llbracket \langle 2200, 02, 28 \rangle = \text{lgc-mjd2grd} ( 124651 ) \rrbracket = \\
\llbracket \langle 2200, 03, 01 \rangle = \text{lgc-mjd2grd} ( 124652 ) \rrbracket = \\
\llbracket \langle 2300, 02, 28 \rangle = \text{lgc-mjd2grd} ( 161175 ) \rrbracket = \\
\llbracket \langle 2300, 03, 01 \rangle = \text{lgc-mjd2grd} ( 161176 ) \rrbracket = \\
\llbracket \langle 2400, 02, 28 \rangle = \text{lgc-mjd2grd} ( 197699 ) \rrbracket = \\
\llbracket \langle 2400, 03, 01 \rangle = \text{lgc-mjd2grd} ( 197701 ) \rrbracket =
\end{aligned}$$

## 10.7 Test of GRD parsing functions

$$\begin{aligned}
\llbracket \text{lgc-convert-leap} ( \langle \text{'GRD-2000-01-02+1'} \rangle ) = \langle \text{lgc-grd2mjd} ( \langle 2000, 01, 02 \rangle ) :: +1 \rangle \rrbracket = \\
\llbracket \text{lgc-convert-leap} ( \langle \text{'GRD-2000-01-02-1'} \rangle ) = \langle \text{lgc-grd2mjd} ( \langle 2000, 01, 02 \rangle ) :: -1 \rangle \rrbracket = \\
\llbracket \text{lgc-convert-leap} ( \langle \text{'GRD-2000-01-02+1'}, \text{'GRD-2000-01-01-1'} \rangle ) = \langle \text{lgc-grd2mjd} \\
( \langle 2000, 01, 02 \rangle ) :: +1, \text{lgc-grd2mjd} ( \langle 2000, 01, 01 \rangle ) :: -1 \rangle \rrbracket =
\end{aligned}$$

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘ERD-2000-01-01-1’⟩ ) = (‘Invalid leap:  
’: : ‘ERD-2000-01-01-1’)•]=

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘GRD:2000-01-01-1’⟩ ) = (‘Invalid leap:  
’: : ‘GRD:2000-01-01-1’)•]=

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘GRD-20A0-01-01-1’⟩ ) = (‘Invalid leap:  
’: : ‘GRD-20A0-01-01-1’)•]=

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘GRD-2000:01-01-1’⟩ ) = (‘Invalid leap:  
’: : ‘GRD-2000:01-01-1’)•]=

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘GRD-2000-01:01-1’⟩ ) = (‘Invalid leap:  
’: : ‘GRD-2000-01:01-1’)•]=

[lgc-convert-leap ( ⟨‘GRD-2000-01-02+1’, ‘GRD-2000-01-01:1’⟩ ) = (‘Invalid leap:  
’: : ‘GRD-2000-01-01:1’)•]=

[lgc-convert-leap ( ⟨‘ARD-2000-01-02+1’, ‘GRD-2000-01-01:1’⟩ ) = (‘Invalid leap:  
’: : ‘ARD-2000-01-02+1’)•]=

[lgc-process-leap ( T[⟨‘parameters’, ‘leap’⟩⇒⟨‘GRD-2000-01-02+1’, ‘GRD-2000-01-01-1’⟩]  
)[‘leap’] = ⟨10, lgc-grd2mjd ( ⟨2000, 01, 03⟩ )·24·60·60+10 : : +1, lgc-grd2mjd  
( ⟨2000, 01, 02⟩ ) · 24 · 60 · 60 + 9 : : -1⟩]=

[lgc-process-leap ( T[⟨‘parameters’, ‘leap’⟩⇒⟨‘GRD-2000-01-02+1’, ‘GRD-2000-01-01-1’⟩]  
)[‘leap’] = ⟨12, lgc-grd2mjd ( ⟨2000, 01, 03⟩ )·24·60·60+12 : : +1, lgc-grd2mjd  
( ⟨2000, 01, 02⟩ ) · 24 · 60 · 60 + 11 : : +1⟩]=

[lgc-process-leap ( T[⟨‘parameters’, ‘leap’⟩⇒⟨‘GRD-2000-01-02+1’, ‘ARD-2000-01-01-1’⟩]  
)[‘leap’] = (‘Invalid leap: ’ : : ‘ARD-2000-01-01-1’)•]=

[lgc-process-leap ( T[⟨‘parameters’, ‘leap’⟩⇒⟨‘GRD-2000-01-01-1’, ‘GRD-2000-01-01-1’⟩]  
)[‘leap’] = ‘Leap seconds must be stated in descending order’•]=

## 10.8 Test of conversion from Logiweb time to MJD/TAI

[lgc-lgt2mjdtai ( ⟨0, 6⟩ ) = ⟨0, 0, 0, 0, 0, 6⟩]=

[lgc-lgt2mjdtai ( ⟨0, 2⟩ ) = ⟨0, 0, 0, 0, 0, 2⟩]=

[lgc-lgt2mjdtai ( ⟨0, 0⟩ ) = ⟨0, 0, 0, 0, 0, 0⟩]=

[lgc-lgt2mjdtai ( ⟨1, 6⟩ ) = ⟨0, 0, 0, 0, 1, 6⟩]=

[lgc-lgt2mjdtai ( ⟨1, 2⟩ ) = ⟨0, 0, 0, 0, 1, 2⟩]=

[lgc-lgt2mjdtai ( ⟨10, 2⟩ ) = ⟨0, 0, 0, 0, 10, 2⟩]=

[lgc-lgt2mjdtai ( ⟨100, 2⟩ ) = ⟨0, 0, 0, 1, 0, 2⟩]=

[lgc-lgt2mjdtai ( ⟨100 · 10, 2⟩ ) = ⟨0, 0, 0, 10, 0, 2⟩]=

$[lgc-lgt2mjd tai ( \langle 100 \cdot 60, 2 \rangle ) = \langle 0, 0, 1, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle 100 \cdot 60 \cdot 10, 2 \rangle ) = \langle 0, 0, 10, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle 100 \cdot 60 \cdot 60, 2 \rangle ) = \langle 0, 1, 0, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle 100 \cdot 60 \cdot 60 \cdot 10, 2 \rangle ) = \langle 0, 10, 0, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle 100 \cdot 60 \cdot 60 \cdot 24, 2 \rangle ) = \langle 1, 0, 0, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle 100 \cdot 60 \cdot 60 \cdot 24 \cdot 10, 2 \rangle ) = \langle 10, 0, 0, 0, 0, 2 \rangle ] =$   
 $[lgc-lgt2mjd tai ( \langle -100 \cdot 60 \cdot 60 \cdot 24 \cdot 10, 2 \rangle ) = \langle -10, 0, 0, 0, 0, 2 \rangle ] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 0, 6 \rangle ) ) = 'MJD-0.TAI:00:00:00.000000'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 0, 2 \rangle ) ) = 'MJD-0.TAI:00:00:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 0, 0 \rangle ) ) = 'MJD-0.TAI:00:00:00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 1, 6 \rangle ) ) = 'MJD-0.TAI:00:00:00.000001'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 1, 2 \rangle ) ) = 'MJD-0.TAI:00:00:00.01'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 10, 2 \rangle ) ) = 'MJD-0.TAI:00:00:00.10'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100, 2 \rangle ) ) = 'MJD-0.TAI:00:00:01.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 10, 2 \rangle ) ) = 'MJD-0.TAI:00:00:10.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60, 2 \rangle ) ) = 'MJD-0.TAI:00:01:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60 \cdot 10, 2 \rangle ) ) = 'MJD-0.TAI:00:10:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60 \cdot 60, 2 \rangle ) ) = 'MJD-0.TAI:01:00:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60 \cdot 60 \cdot 10, 2 \rangle ) ) = 'MJD-0.TAI:10:00:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60 \cdot 60 \cdot 24, 2 \rangle ) ) = 'MJD-1.TAI:00:00:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle 100 \cdot 60 \cdot 60 \cdot 24 \cdot 10, 2 \rangle ) ) = 'MJD-10.TAI:00:00:00.00'] =$   
 $[vt2vector ( lgc-lgt2mjd tai2vt ( \langle -100 \cdot 60 \cdot 60 \cdot 24 \cdot 10, 2 \rangle ) ) = 'MJD--10.TAI:00:00:00.00'] =$

## 10.9 Test of conversion from Logiweb time to GRD/UTC

[test-leaps  $\equiv$

$\langle 11, 6 \cdot 60 + 11 :: +1, 4 \cdot 60 + 10 :: +1, 2 \cdot 60 + 9 :: -1 \rangle$

A situation where a negative leap occurred two minutes after epoch and two positive ones occurred four and six minutes after epoch.

$[0 :: 0 = lgc-lgt2utc ( 0 , 0 , \langle \rangle ) ] =$

$[-10::0 = \text{lgc-lgt2utc} ( 0 , 10 , \langle \rangle )]=$   
 $[0::0 = \text{lgc-lgt2utc} ( 10 , 10 , \langle \rangle )]=$   
 $[362::0 = \text{lgc-lgt2utc} ( 373 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[361::0 = \text{lgc-lgt2utc} ( 372 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[360::0 = \text{lgc-lgt2utc} ( 371 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[359::1 = \text{lgc-lgt2utc} ( 370 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[359::0 = \text{lgc-lgt2utc} ( 369 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[358::0 = \text{lgc-lgt2utc} ( 368 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[302::0 = \text{lgc-lgt2utc} ( 312 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[301::0 = \text{lgc-lgt2utc} ( 311 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[300::0 = \text{lgc-lgt2utc} ( 310 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[299::0 = \text{lgc-lgt2utc} ( 309 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[298::0 = \text{lgc-lgt2utc} ( 308 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[297::0 = \text{lgc-lgt2utc} ( 307 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[242::0 = \text{lgc-lgt2utc} ( 252 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[241::0 = \text{lgc-lgt2utc} ( 251 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[240::0 = \text{lgc-lgt2utc} ( 250 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[239::1 = \text{lgc-lgt2utc} ( 249 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[239::0 = \text{lgc-lgt2utc} ( 248 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[238::0 = \text{lgc-lgt2utc} ( 247 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[182::0 = \text{lgc-lgt2utc} ( 191 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[181::0 = \text{lgc-lgt2utc} ( 190 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[180::0 = \text{lgc-lgt2utc} ( 189 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[179::0 = \text{lgc-lgt2utc} ( 188 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[178::0 = \text{lgc-lgt2utc} ( 187 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[177::0 = \text{lgc-lgt2utc} ( 186 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[122::0 = \text{lgc-lgt2utc} ( 131 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[121::0 = \text{lgc-lgt2utc} ( 130 , \text{test-leaps}^h , \text{test-leaps}^t )]=$

$[120::0 = \text{lgc-lgt2utc} ( 129 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[118::0 = \text{lgc-lgt2utc} ( 128 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[117::0 = \text{lgc-lgt2utc} ( 127 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[116::0 = \text{lgc-lgt2utc} ( 126 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[62::0 = \text{lgc-lgt2utc} ( 72 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[61::0 = \text{lgc-lgt2utc} ( 71 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[60::0 = \text{lgc-lgt2utc} ( 70 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[59::0 = \text{lgc-lgt2utc} ( 69 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[58::0 = \text{lgc-lgt2utc} ( 68 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[57::0 = \text{lgc-lgt2utc} ( 67 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[2::0 = \text{lgc-lgt2utc} ( 12 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[1::0 = \text{lgc-lgt2utc} ( 11 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[0::0 = \text{lgc-lgt2utc} ( 10 , \text{test-leaps}^h , \text{test-leaps}^t )]=$   
 $[\text{test-leapstate} \stackrel{\bullet\bullet}{\equiv} \top[\text{'leap'} \rightarrow \text{test-leaps}]]$   
 $[[1858, 11, 17, 0, 0, 0, 0, 6] = \text{lgc-lgt2grdutc} ( \langle 0, 6 \rangle , \top[\text{'leap'} \rightarrow \langle 0 \rangle] )]=$   
 $[[1858, 11, 17, 0, 0, 10, 0, 6] = \text{lgc-lgt2grdutc} ( \langle 0, 6 \rangle , \top[\text{'leap'} \rightarrow \langle -10 \rangle] )]=$   
 $[[1858, 11, 17, 0, 0, 11, 23, 2] = \text{lgc-lgt2grdutc} ( \langle 123, 2 \rangle , \top[\text{'leap'} \rightarrow \langle -10 \rangle] )]=$   
 $[[1858, 11, 17, 0, 0, 11, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 1, 0 \rangle , \top[\text{'leap'} \rightarrow \langle -10 \rangle] )]=$   
 $[[1858, 11, 17, 0, 0, 00, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 10, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 0, 01, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 11, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 0, 02, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 12, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 0, 57, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 67, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 0, 58, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 68, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 0, 59, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 69, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 1, 00, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 70, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 1, 01, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 71, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 1, 02, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 72, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $[[1858, 11, 17, 0, 1, 56, 0, 0] = \text{lgc-lgt2grdutc} ( \langle 126, 0, 0 \rangle , \text{test-leapstate} )]=$



$[(1858, 11, 17, 0, 6, 02, 0, 0) = \text{lgc-lgt2grdutc} ( \langle 373, 0, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:00.000000}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 0, 6 \rangle , \text{T}['\text{leap}' \rightarrow \langle 0 \rangle] )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:10.000000}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 0, 6 \rangle , \text{T}['\text{leap}' \rightarrow \langle -10 \rangle] )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:11.23}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 123, 2 \rangle , \text{T}['\text{leap}' \rightarrow \langle -10 \rangle] )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:11}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 1, 0 \rangle , \text{T}['\text{leap}' \rightarrow \langle -10 \rangle] )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:00}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 10, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:01}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 11, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:02}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 12, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:57}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 67, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:58}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 68, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:00:59}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 69, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:00}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 70, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:01}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 71, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:02}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 72, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:56}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 126, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:57}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 127, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:01:58}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 128, 0 \rangle , \text{test-leapstate} )]=$   
 $['\text{GRD-1858-11-17.UTC:00:02:00}' =$   
 $\text{lgc-lgt2grdutc2v} ( \langle 129, 0 \rangle , \text{test-leapstate} )]=$

['GRD-1858-11-17.UTC:00:02:01' =  
     lgc-lgt2grdutc2v ( ⟨130, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:02:02' =  
     lgc-lgt2grdutc2v ( ⟨131, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:02:57' =  
     lgc-lgt2grdutc2v ( ⟨186, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:02:58' =  
     lgc-lgt2grdutc2v ( ⟨187, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:02:59' =  
     lgc-lgt2grdutc2v ( ⟨188, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:00' =  
     lgc-lgt2grdutc2v ( ⟨189, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:01' =  
     lgc-lgt2grdutc2v ( ⟨190, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:02' =  
     lgc-lgt2grdutc2v ( ⟨191, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:58' =  
     lgc-lgt2grdutc2v ( ⟨247, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:59' =  
     lgc-lgt2grdutc2v ( ⟨248, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:03:60' =  
     lgc-lgt2grdutc2v ( ⟨249, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:00' =  
     lgc-lgt2grdutc2v ( ⟨250, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:01' =  
     lgc-lgt2grdutc2v ( ⟨251, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:02' =  
     lgc-lgt2grdutc2v ( ⟨252, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:57' =  
     lgc-lgt2grdutc2v ( ⟨307, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:58' =  
     lgc-lgt2grdutc2v ( ⟨308, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:04:59' =  
     lgc-lgt2grdutc2v ( ⟨309, 0⟩ , test-leapstate )]=

['GRD-1858-11-17.UTC:00:05:00' =  
     lgc-lgt2grdutc2v ( ⟨310, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:05:01' =  
     lgc-lgt2grdutc2v ( ⟨311, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:05:02' =  
     lgc-lgt2grdutc2v ( ⟨312, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:05:58' =  
     lgc-lgt2grdutc2v ( ⟨368, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:05:59' =  
     lgc-lgt2grdutc2v ( ⟨369, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:05:60' =  
     lgc-lgt2grdutc2v ( ⟨370, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:06:00' =  
     lgc-lgt2grdutc2v ( ⟨371, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:06:01' =  
     lgc-lgt2grdutc2v ( ⟨372, 0⟩ , test-leapstate )]=  
 ['GRD-1858-11-17.UTC:00:06:02' =  
     lgc-lgt2grdutc2v ( ⟨373, 0⟩ , test-leapstate )]=  
 [test-leapstate2  $\equiv$   
     lgc-process-leap ( T[⟨'parameters', 'leap'⟩⇒lgc-default-leap] )]  
 ['GRD-2009-02-18.UTC:08:56:04' =  
     lgc-lgt2grdutc2v ( ⟨4741664198, 0⟩ , test-leapstate2 )]=  
     GRD-2009-02-18.UTC:08:56:04 = MJD-54880.TAI:08:56:38.  
 ['GRD-1858-11-17.UTC:00:00:00' =  
     lgc-lgt2grdutc2v ( ⟨10, 0⟩ , test-leapstate2 )]=  
 ['GRD-1972-07-01.UTC:00:00:01' =  
     lgc-lgt2grdutc2v ( ⟨lgc-grd2mjd ( ⟨1972, 7, 1⟩ ) · 24 · 60 · 60 + 12, 0⟩ ,  
     test-leapstate2 )]=  
 ['GRD-1972-07-01.UTC:00:00:00' =  
     lgc-lgt2grdutc2v ( ⟨lgc-grd2mjd ( ⟨1972, 7, 1⟩ ) · 24 · 60 · 60 + 11, 0⟩ ,  
     test-leapstate2 )]=  
 ['GRD-1972-06-30.UTC:23:59:60' =  
     lgc-lgt2grdutc2v ( ⟨lgc-grd2mjd ( ⟨1972, 7, 1⟩ ) · 24 · 60 · 60 + 10, 0⟩ ,  
     test-leapstate2 )]=

['GRD-1972-06-30.UTC:23:59:59' =  
lgc-lgt2grdutc2v ( ⟨lgc-grd2mjd ( ⟨1972, 7, 1⟩ ) · 24 · 60 · 60 + 9, 0⟩ ,  
test-leapstate2 ) ] =

['GRD-1972-06-30.UTC:23:59:58' =  
lgc-lgt2grdutc2v ( ⟨lgc-grd2mjd ( ⟨1972, 7, 1⟩ ) · 24 · 60 · 60 + 8, 0⟩ ,  
test-leapstate2 ) ] =

## 10.10 Test of conversion from Unix time to Logiweb time

['GRD-1970-01-01.UTC:00:00:00' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨0, 0⟩ , test-leapstate2 ) , test-leapstate2 ) ] =

['GRD-1970-01-01.UTC:00:00:00.000000' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨0, 6⟩ , test-leapstate2 ) , test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:58.0' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487980, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:58.9' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487989, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:59.0' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487990, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:59.2' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487991, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:59.4' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487992, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:59.6' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487993, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:59.8' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487994, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

['GRD-1998-12-31.UTC:23:59:60.0' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487995, 1⟩ , test-leapstate2 ) ,  
test-leapstate2 ) ] =

```

['GRD-1998-12-31.UTC:23:59:60.2' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487996, 1⟩ , test-leapstate2 )
    , test-leapstate2 )]=

['GRD-1998-12-31.UTC:23:59:60.4' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487997, 1⟩ , test-leapstate2 )
    , test-leapstate2 )]=

['GRD-1998-12-31.UTC:23:59:60.6' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487998, 1⟩ , test-leapstate2 )
    , test-leapstate2 )]=

['GRD-1998-12-31.UTC:23:59:60.8' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151487999, 1⟩ , test-leapstate2 )
    , test-leapstate2 )]=

['GRD-1999-01-01.UTC:00:00:00.0' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨9151488000, 1⟩ , test-leapstate2 )
    , test-leapstate2 )]=

[test-leapstate3 ≐
  lgc-process-leap ( T[⟨'parameters', 'leap'⟩⇒⟨'GRD-1970-01-01-1'⟩] ) ]

['GRD-1970-01-01.UTC:00:00:00' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨0, 0⟩ , test-leapstate3 ) , test-leapstate3
    )]=

['GRD-1970-01-01.UTC:00:00:00.000000' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨0, 6⟩ , test-leapstate3 ) , test-leapstate3
    )]=

['GRD-1970-01-01.UTC:23:59:57.0' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨863970, 1⟩ , test-leapstate3 ) ,
    test-leapstate3 )]=

['GRD-1970-01-01.UTC:23:59:57.9' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨863979, 1⟩ , test-leapstate3 ) ,
    test-leapstate3 )]=

['GRD-1970-01-01.UTC:23:59:58.00' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨863980, 1⟩ , test-leapstate3 ) ,
    test-leapstate3 )]=

['GRD-1970-01-01.UTC:23:59:58.05' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨863981, 1⟩ , test-leapstate3 ) ,
    test-leapstate3 )]=

['GRD-1970-01-01.UTC:23:59:58.45' =
  lgc-lgt2grdutc2v ( lgc-unix2lgt ( ⟨863989, 1⟩ , test-leapstate3 ) ,
    test-leapstate3 )]=

```

['GRD-1970-01-01.UTC:23:59:58.50' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( <863990, 1> , test-leapstate3 ) ,  
test-leapstate3 )]=

['GRD-1970-01-01.UTC:23:59:58.95' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( <863999, 1> , test-leapstate3 ) ,  
test-leapstate3 )]=

['GRD-1970-01-02.UTC:00:00:00.0' =  
lgc-lgt2grdutc2v ( lgc-unix2lgt ( <864000, 1> , test-leapstate3 ) ,  
test-leapstate3 )]=

## References

- [1] Klaus Grue. Compiler - appendix. Technical report, Logiweb, 2008. [../..../logiweb/010B8CEF872E4CAE653DCBA950DA8437DE154147ADBAE6B1F9D5B2BB0806/page/appendix.pdf](http://logiweb/010B8CEF872E4CAE653DCBA950DA8437DE154147ADBAE6B1F9D5B2BB0806/page/appendix.pdf).