# TAME Manual v4.0.4-45-gc54a87e

Ryan Specialty Group

This manual is for TAME, version 4.0.4-45-gc54a87e.

*This manual contains inline notes for developers of TAME.*[1] *For an index of notes, see* *[Developer Notes Index], page 32**.*

[1] To disable for user documentation, pass `--disable-devnotes` to `configure`.

# Table of Contents

# 1 Using TAME

TAME is The Adaptive Metalanguage, a programming language and system of tools designed to aid in the development; understanding; and maintenance of systems performing numerous calculations on a complex graph of dependencies; conditions; and a large number of inputs.

This system was developed at Ryan Specialty Group[1] to handle the complexity of comparative insurance rating systems. It is a domain-specific language (DSL) that itself encourages, through the use of templates, the creation of sub-DSLs. TAME itself is at heart a calculator— processing only numerical input and output— driven by quantifiers as predicates. Calculations and quantifiers are written declaratively without concern for order of execution.

The system has powerful dependency resolution and data flow capabilities.

TAME consists of a macro processor (implementing a metalanguage); numerous compilers for various targets (JavaScript, HTML documentation and debugging environment, LaTeX, and others); linkers; and supporting tools. The input grammar is XML, and the majority of the project (including the macro processor, compilers, and linkers) is written in XSLT.[2]

## 1.1 Getting Started

To get started, make sure Saxon version 9 or later is available and its path set as *SAXON_CP*; that the path to hoxsl is set via *HOXSL*; and then run the '`bootstrap`' script:

```
$ export SAXON_CP=/path/to/saxon9he.jar
$ export HOXSL=/path/to/hoxsl/root

$ ./boostrap
```

Figure 1.1: Bootstrapping TAME in a source repository checkout

## 1.2 Manual Compilation

Note: TAME is usually controlled through a Makefile; see Section 1.3 [Compiling With Make], page 3 to avoid manual compilation steps.

TAME is controlled through the program in `bin/tame`. When run, it first spawns a daemon `bin/tamed` if it is not already running. `tamed` is needed to keep the JVM and compiled XSLT templates in memory, otherwise each file would incur a steep startup penalty.

*TODO: Document commands. Most developers do not build files directly, so this is not essential yet.*

---

[1] Formerly LoVullo Associates.

[2] There is a reason for that odd choice; until an explanation is provided, know that someone is perverted enough to implement a full compiler in XSLT.

```
$ bin/tame compile src/foo.xml src/foo.xmlo
$ bin/tame link src/foo.xmlo src/foo.xmle
$ bin/tame standalone src/foo.xmle src/foo.js
$ bin/tame summary src/foo.xmle src/foo.html
```

Figure 1.2: Compiling a JavaScript executable and Summary Page

To kill the daemon, pass '`--kill`' to either `bin/tame` or `bin/tamed`. For additional options and environment variables that influence operation, pass '`--help`' to either command.

## 1.3 Compiling With Make

TAME can generate a GNU Makefile for you using Automake and Autoconf. This greatly simplifies building projects by automatically building all dependencies as needed, and only when they have changed.[3]

The Makefile is generated by a `configure` script, which itself generated by Autoconf using `configure.ac` in the project root:

```
AC_INIT([PROJECT_NAME], [0.0.0], [contact@email])

m4_define(‘calc_root’, rater)
m4_include([rater/build-aux/m4/calcdsl.m4])
```

Figure 1.3: Example `configure.ac` in project root.

By convention, TAME is usually available as a submodule under `rater/`. This confusing naming convention may change in the future.

Then, to generate the `Makefile`:

```
$ autoreconf -fvi
$ ./configure SAXON_CP=/path/to/saxon9he.jar
```

Figure 1.4: Invoking `configure` to generate `Makefile`.

*TODO: Add more sections.*

### 1.3.1 Common Targets

A *target* is something that can be built. Usually it is a specific file (e.g. `foo.js`), but it can also be a command (also called a *phony target*). Here are the most common phony targets that may be useful:

'`all`' This is the default target (just type '`make`'). Build the UI and all suppliers. Does not build the Summary Pages, as they are considered to be debugging tools.

'`summary-html`'
Build all Summary Pages for programs in `suppliers/`. This is equivalent to building each `suppliers/*.html` target manually.

[3] When their modification timestamps change, specifically.

‘`check`’

‘`test`’ Run test cases in `test/`.

‘`standalones`’
Build JavaScript executables for each program in `suppliers/`. This is a dependency of ‘`summary-html`’.

‘`tamed-die`’
‘`kill-tamed`’
Kill running tamed for effective user, if any (see Section 1.2 [Manual Compilation], page 2).

‘`clean`’ Delete all file targets. This may be necessary when upgrading TAME, for example, to rebuild all files using the new version.

### 1.3.2 Parallel Builds

GNU Make offers parallel builds through the `-j` flag, which specifies the maximum number of concurrent jobs. This is supported by both `tame` and `tamed`.

`tamed` starts by spawning a single runner, which is marked as available. When a command is issued to `tame`, it will reserve the first available runner it finds by marking it as busy. Once the runner is finished, it will be marked as available once again. If all available runners are busy, `tame` issues a signal to `tamed` to spawn another runner, which `tame` then reserves and marks as busy. No runners are ever freed (terminated) until `tamed` itself terminates.

For example, to build with up to four concurrent runners, use ‘`-j4`’:

```
$ make -j4
```

Figure 1.5: Compiling with four concurrent runners

Compiling and linking large packages can be memory intensive. While runner memory consumption may vary, it’s wise to profile the memory usage of a single runner and use that to estimate how many concurrent runners your system can support.

Saxon is also multi-threaded under certain circumstances, so you should allocate fewer jobs than you have available CPU cores. GNU Make also offers a `-l` flag that tells it not to spawn more jobs if the system is above the indicated load. But note that, even if a runner is idle, it is still using up memory.

# 2 Core Concepts

*There isn't much here yet. Maybe you can help?*

TAMEis a *declarative* programming language— it *describes* how a program ought to behave rather than explicitly telling a computer the steps needed to make it work (*imperative*). TAMEis a calculator at heart, so code written in its language describes mathematical operations. Definitions fall primarily under two categories:

**Classifications**
: Higher-order logic used to classify data and predicate calculations, controlling program flow.

**Calculations**
: Mathematical operations motivated by linear algebra.

TAMEalso supports abstracting calculations into *functions*, which permits the use of recursion for solving problems that are difficult to fit into an algebraic model. The language is Turing-complete.

TAMEitself is a *domain-specific language* (DSL)— it was designed for use in comparative insurance rating systems. However, it couldn't possibly know all useful abstractions that may be needed in the future; the domain for which TAMEwas designed encompasses many subdomains as well. To accommodate future needs, TAMEis also a *metalanguage*— a language that can be used to program itself. The core language is based upon broad mathematical foundations that offer great flexibility. Its expressive template system allows the creation of abstractions that are indistinguishable from core language features, and templates have powerful introspective capabilities that allow for many useful types of code generation. Essentially, TAMEallows the creation of sub-DSLs.

Code is written in TAMEwithout regard to order of execution— the linker will figure that out for you. This simplifies the development of systems with complex graphs of dependencies.

*TODO: This chapter is a placeholder. More to come.*

# Appendix A Preprocessor

## A.1 Macro Expansion

### A.1.1 Expansion Sequence

An *expansion sequence* $E$ is an ordered list of nodes $N_1, N_2, \ldots, N_m$ satisfying the property that, given some node $N_x \in E$ such that $m \geq x > 1$, the node $N_{\{x-1\}}$ must have already been fully expanded before expansion of $N_x$ begins. Such an ordering guarantee is generally unnecessary.

Expansion sequences are initiated by invoking [eseq:expand-step#1], page 6 on any arbitrary node containing any number of children to be expanded in order. Each call will proceed one step (detailed herein), eventually resulting in each node expanded and the expansion sequence node eliminated.

`node()* eseq:expand-step` (*eseq as node()**) [function]

`xmlns:eseq="http://www.lovullo.com/tame/preproc/expand/eseq"`

This fuction performs only a single pass, expected to be applied once for each pass as necessary. This may change in the future once the preprocessor supports subtree expansion.

The final result of the expansion is the result of expanding each child serially, requiring that expansion of the previous sibling be wholly completed before expansion of a node. The parent expansion sequence node will be stripped from the result once empty.

Note that this function accepts an empty XML sequence, and will return an empty sequence in such a case.

*Definition:*

```
<function name="eseq:expand-step"  as="node()*">
  <param name="eseq"  as="node()*" />

  <variable name="count"  as="xs:integer"  select="count( $eseq )" />

  <variable name="target"  as="element()?"  select="$eseq[ $count ]" />

  <!-- We must be careful in how we retain other nodes; since we
       accept a sequence of nodes, the @code{preceding-sibling}
       pseudo-selector cannot be used, since they are not part of the
       same tree. -->
  <sequence select="$eseq[ position() lt $count ]" />

  <apply-templates mode="_eseq:expand"  select="$target" />
</function>
```

Throughout the process, we will consider only the head of the sequence for processing; we call the the *head node*. Once the head is done expanding, it is *hoisted* out out of the sequence—order maintained—and processing continues with the next head. If no such head exists, then we are done.

This implementation sounds simple on the face of it, but is complicated by the fact that we are not performing this operation in one sweep—due to the implementation of the preprocessor at the time that this was written, we must pass control back with each pass that might trigger an expansion, allowing the symbol table to be updated and other processing to take place. At least that's the rationale; we wouldn't want to make such assumptions in this implementation. But we do need to keep that in mind.

With that, we will also want to keep the number of re-passes to a minimum; that means yielding back to the caller only when necessary. Until a redesign of the preprocessor such that it will reprocess only as necessary, running a sequential expansion will *always* be slower, and processing time will grow linearly with the number of expansion nodes, relative to the size of the entire package.

#### A.1.1.1 Predicating Expansion

To start, let's consider possible nodes. We do not want to make any assumptions of what may or may not be expandable, so we will handle them all consistently. The first question is whether the node is even expandable.

Our perspective is abstraction—we do not know (nor should we) what the caller may consider to be expandable, or how it may be expanded.

To solve this problem, we introduce an abstract predicate that must be overridden by an implementation to provide anything of value. Otherwise, all we can do is assume that the node we have encountered cannot be expanded (what would we even do to expand it?).

`xs:boolean eseq:is-expandable` (*node as node*()) [function]

`xmlns:eseq="http://www.lovullo.com/tame/preproc/expand/eseq"`

*An implementation must override this function.*

This predicate determines exclusively whether a node should be expanded or hoisted. Therefore, it should account for both nodes that *cannot* be expanded (for example, text nodes may not be expanded), and nodes that have *already been expanded* that can undergo no further expansion.

The default implementation therefore will always yield false, meaning that no processing will take place.

*Definition:*

```
<function name="eseq:is-expandable"  as="xs:boolean">
  <param name="node"  as="node()" />

  <sequence select="false()" />
</function>
```

With that question answered, we are now able to proceed:

1. If a node is not expandable, we can immediately hoist it (see Section A.1.1.3 [Node Hoisting], page 11);
2. Otherwise, we must allow the node to attempt to expand.

The former allows us to save re-passes (and therefore improve performance), as well as the headache of handling every possible case.[1] The latter has an important consideration.

[1] Well, we deferred that complexity to the caller via our `eseq:is-expandable` predicate.

Let's start with the first case.

`_eseq:expand` *on* *[ *node*()[*1*][ *not*( *eseq:is-expandable*( . ) ) ] ] [match]

When we encounter a head that is not expandable, it will be immediately hoisted, as there is no work to be done.

The result of this operation will be a sequence of nodes, one of them being the hoisted node, and the last being the remaining expansion sequence. See Section A.1.1.3 [Node Hoisting], page 11.

```
<template mode="_eseq:expand"  as="node()+"  match="*[ node()[1][ not( eseq:is-expanda
  <sequence select="_eseq:hoist( . )" />
</template>
```

As we only have an abstract view of the concept of "expansion", we need a way for an implementation to notify us whether a node needs further expansion, or is ready to be hoisted. Fortunately, we already have that information because of how we defined [eseq:is-expandable#1], page 7. In other words, our previously declared processing will already take care of hoisting for us when necessary, so we need only continue to expand nodes as necessary.

`_eseq:expand` *on* *[ *node*()[*1*][ *eseq:is-expandable*( . ) ] ] [match]

We must continue to expand expandable nodes; otherwise, nested expansions would never take place.

Once expansion is complete, by the definition of [eseq:is-expandable#1], page 7, expansion will halt.

```
<template mode="_eseq:expand"  as="element()"  match="*[ node()[1][ eseq:is-expandable
  <sequence select="_eseq:expand-head( . )" />
</template>
```

Up until this point, we have been assuming that there is an actual head node to process; this may not be the case. In fact, we are guaranteed to encounter and empty expansion sequence at some point, because all nodes will have been hoisted (see Section A.1.1.3 [Node Hoisting], page 11)!

In this instance, we will consider our job to have been completed, and self-destruct.

`_eseq:expand` *on* *[ *not*( *node*() ) ] [match]

Once there is no head node, expansion is complete and the sequence parent is no longer necessary.

```
<template mode="_eseq:expand"  match="*[ not( node() ) ]" />
```

The above matches satisfy all possible conditions.

Proof: Let the expansion sequence element be the context node. As it is an element, it is matched by `*`, which is the context of each match. The expansion sequence either has a head node, or it does not have a head node. If it *does* have a head node, then it can be defined as `node()[1]`; otherwise `node()` yields an empty sequence, and the final template is matched. When the head node *is* available, it is either expandable or non-expandable, determined by the predicate [eseq:is-expandable#1], page 7. Since the predicate returns a boolean, it must be either `false()` or `true()`, and so it must satisfy either the first or second template respectively.

We have therefore determined hoisting/expansion actions through use of a single predicate.

An astute reader may have come to an uncomfortable realization: after all expansions are complete and the expansion sequence node itself is eliminated (per the final match above), then the node that was last expanded and hoisted will be considered to be the expansion sequence by [eseq:expand-step#1], page 6. This is true, but should not be a problem in practice: hoisting is intended to place nodes into context for the caller; it is expected that the caller will recognize when to invoke sequence expansion (likely on a pre-defined node type, which would no longer match after it is eliminated). The discomfort comes from the fact that we cannot use this implementation recursively; this is a consequence of the current preprocessor implementation, and is subject to change in the future.

### A.1.1.2 Expansion Operations

The concept of "expansion" is treated as an abstract operation (see Section A.1.1.1 [Predicating Expansion], page 7). The system implementing expansion sequences should provide concrete definitions of what it means to expand a node, and what it means to check whether a node is expanded.

Recall that expansion takes place only the on the head node. Expanding the head therefore involves reproducing the expansion sequence with head expanded and siblings untouched.

`element() _eseq:expand-head` (*eseq as element*()) [function]
`xmlns:_eseq="http://www.lovullo.com/tame/preproc/expand/eseq/_priv"`

Expanding the head produces a new expansion sequence with the head expanded and all its sibling nodes left untouched. This produces the fundamental effect of the expansion sequence.

A head node *must* be available to satisfy the domain of the function.

Actual expansion is left to [eseq:expand-node#1], page 10.

*Definition:*

```
<function name="_eseq:expand-head"  as="element()">
  <param name="eseq"  as="element()" />

  <variable name="head"  as="node()"  select="$eseq/node()[1]" />

  <!-- This @code{for-each} is purely to set the context for
       @code{copy}, since we do not know the sequence element
       name. -->
  <for-each select="$eseq">
    <copy>
      <sequence select="$eseq/@*, eseq:expand-node( $head ), $head/following-sibling::
    </copy>
  </for-each>
</function>
```

Just as we defined an overridable expansion predicate, we will provide an overridable function that performs actual node expansion.

Its default behavior is an important consideration: what if [eseq:is-expandable#1], page 7 is overridden but the implementation forgets to override [eseq:expand-node#1], page 10? If the default behavior were to simply echo back the node, it seems likely that we would never finish processing, since the very node that matched the predicate to begin with would remain unchanged.

Ideally, we remain side-effect free (meaning no triggering of errors). Instead, we return a single node in our own namespace that represents the error; this will likely trigger an error in the system, since the node is unrecognized.[2]

`node()* eseq:expand-node` (*node as node*()) [function]

`xmlns:eseq="http://www.lovullo.com/tame/preproc/expand/eseq"`

*An implementation must override this function.*

This function should perform whatever is necessary to expand the provided node. Note that this call represents a single step in an expansion, so it need not result in a complete expansion; further processing will take place according to the result of the [eseq:is-expandable#1], page 7 predicate.

If [eseq:is-expandable#1], page 7 is provided, but an override for this function is not, then the default behavior is to return a node in our namespace providing a description of the problem; this is to prevent infinite recursion/iteration.

*Definition:*

```
<function name="eseq:expand-node"  as="node()*">
  <param name="node"  as="node()" />

  <eseq:expand-error>
    <text>A node expansion was requested via ‘eseq:expand-node’</text>
    <text>, but no implementation was provided.  Please</text>
    <text>override the function.</text>
  </eseq:expand-error>
</function>
```

The return type of [eseq:expand-node#1], page 10 produces an interesting concept. Consider what may happen after an expansion:

1. Node expanded into a single node;
2. Node expanded into nothing (producing no node); or
3. Node expanded into multiple nodes.

The first is obviously not an issue, since it keeps us consistent with what we have been doing. In the second case, a node is removed rather than being hoisted, but we are otherwise in a state that we expect: less a node. So will the case of expanding into multiple nodes cause any problems?

It shouldn't, but it is worth a discussion to rationalize. Expansion sequences exist to provide expansion guarantees; the system otherwise expands nodes as it can in an undefined

[2] If an error is *not* triggered, then the system is not very sound.

manner. Since that manner is undefined, providing it with stronger restrictions is acceptable: the newly expanded nodes will be processed in order as a consequence of becoming part of the expansion sequence, but they will otherwise be processed as normal.[3]

After expansion, with the current preprocessor design, we have no choice but the yield control back to the caller to allow it to continue processing; the expansion may have yielded additional symbols that must be added to the symbol table, for example. The process will be continued on the next call to [eseq:expand-step#1], page 6.

### A.1.1.3 Node Hoisting

*Hoisting* is the process of moving a fully expanded head node out of the expansion sequence; it is the final step of the process for a head node and is driven wholly by the [eseq:is-expandable#1], page 7 predicate.

Unfortunately, we cannot continue processing immediately after hoisting for the same reasons that we cannot continue processing after expansion: after hoisting, the nodes may enter a proper context and acquire another meaning, which may result in, for example, additional symbols.

`node()+ _eseq:hoist` (*eseq-node as element*()) [function]

`xmlns:_eseq="http://www.lovullo.com/tame/preproc/expand/eseq/_priv"`

Hoisting removes the head node from the expansion sequence, leaving all other expansion nodes untouched. The result is a sequence of two nodes, the last of which is the expansion sequence element.

If no head node exists, the result is the single expansion sequence node unchanged.

*Definition:*

```
<function name="_eseq:hoist"  as="node()+">
  <param name="eseq-node"  as="element()" />

  <variable name="head"  as="node()?"  select="$eseq-node/node()[1]" />

  <sequence select="$head" />

  <!-- This @code{for-each} is purely to set the context for
       @code{copy}, since we do not know the sequence element
       name. -->
  <for-each select="$eseq-node">
    <copy>
      <sequence select="$eseq-node/@* , $head/following-sibling::node()" />
    </copy>
  </for-each>
</function>
```

[3] In fact, this is a useful property, since it can be exploited by templates to create abstractions with ordering guarantees. So consider it a feature!

# Appendix B Dependency Graph

The dependency graph is a directed graph consisting of every known symbol, post-expansion (see Section A.1 [Macro Expansion], page 6). Cycles are produced only by function recursion and otherwise cause an error, so the graph is studied as a DAG (directed acyclic graph) with few exceptions.

Vertices in the dependency graph are represented by `preproc:sym-dep` nodes, and edges by child `preproc:sym-ref` nodes. Graphs are represented by `preproc:sym-deps`. The graph of each package is considered to be a subgraph of the entire dependency graph for a particular system.[1]

`element( preproc:sym-deps ) graph:make-from-vertices` [function]
(*vertices as element*( *preproc:sym-dep* )*)
`xmlns:graph="http://www.lovullo.com/tame/graph"`

Create a graph from the given vertex set *$vertices*. The resulting graph will be normalized with duplicate vertices and edges removed, making it suitable for ad hoc graph generation.[2]

*Definition:*

```
<function name="graph:make-from-vertices"  as="element( preproc:sym-deps )">
  <param name="vertices"  as="element( preproc:sym-dep )*" />

  <variable name="graph"  as="element( preproc:sym-deps )">
    <preproc:sym-deps>
        <sequence select="$vertices" />
    </preproc:sym-deps>
  </variable>

  <!-- dedupe/normalize -->
  <sequence select="graph:union( $graph )" />
</function>
```

`element( preproc:sym-deps ) graph:reverse` (*graph as element*( [function]
*preproc:sym-deps* ))
`xmlns:graph="http://www.lovullo.com/tame/graph"`

Produce a new graph that is the transpose of *$graph*—that is, the graph *$graph* with the direction of all of its edges reversed.

This is useful for processing what symbols are *used by* other symbols.

For example:

[1] The node names are for compatibility with legacy systems and may change in the future; always use the graph API, and only use the node QNames for type checks.

[2] This is done by calling [graph:union#1], page 13.

```
G:  A->B->C->E
     \
      `->D

G': A<-B<-C<-E
    ^
     `D
```

Figure B.1: G' is the transpose of G

Edge attributes (`preproc:sym-ref/@*`) will be set to the union of all attributes on all edges of the same `@name` in `$graph`. *If edge attributes do not share the same value, the behavior is undefined.*

*Definition:*

```
<function name="graph:reverse"  as="element( preproc:sym-deps )">
  <param name="graph"  as="element( preproc:sym-deps )" />

  <variable name="reversed"  as="element( preproc:sym-dep )*">
    <for-each-group select="$graph//preproc:sym-ref"  group-by="@name">
      <preproc:sym-dep name="{@name}">
        <for-each select="current-group()/ancestor::preproc:sym-dep">
          <preproc:sym-ref>
            <sequence select="current-group()/@*" />

            <!-- keep our name (overrides the above) -->
            <attribute name="name"  select="@name" />
          </preproc:sym-ref>
        </for-each>
      </preproc:sym-dep>
    </for-each-group>
  </variable>

  <preproc:sym-deps>
    <!-- vertices in $graph with no dependencies will not be in
         $reversed -->
    <for-each select="$graph/preproc:sym-dep[ not( @name = $reversed/preproc:sym-ref/@
      <preproc:sym-dep name="{@name}" />
    </for-each>

    <sequence select="$reversed" />
  </preproc:sym-deps>
</function>
```

`element( preproc:sym-deps )* graph:union` (*graphs as element*( *preproc:sym-deps* )*) [function]

`xmlns:graph="http://www.lovullo.com/tame/graph"`

Merge sequence of graphs *$graphs* into a single graph by taking the union of all vertices and edges. Directionality will be preserved.

Edge attributes (`preproc:sym-ref/@*`) will be set to the union of all attributes on all edges of the same `@name`. *If edge attributes do not share the same value, the behavior is undefined.*

For example:

```
G:  A->B->C
G:  C->A
G:  B->C->D

G:  A->B->C->D
     ^____/
```

Figure B.2: (G G G)

This function also removes duplicate vertices and edges, so it can be used with a single (or multiple) graphs to normalize and tidy things up. Any unknown XML nodes are removed.

*Definition:*

```
<function name="graph:union"  as="element( preproc:sym-deps )*">
  <param name="graphs"  as="element( preproc:sym-deps )*" />

  <preproc:sym-deps>
    <for-each-group select="$graphs/preproc:sym-dep"  group-by="@name">
      <preproc:sym-dep name="{@name}">
        <for-each-group select="current-group()/preproc:sym-ref"  group-by="@name">
          <preproc:sym-ref>
            <sequence select="current-group()/@*" />

            <!-- keep our name (overrides the above) -->
            <attribute name="name"  select="@name" />
          </preproc:sym-ref>
        </for-each-group>
      </preproc:sym-dep>
    </for-each-group>
  </preproc:sym-deps>
</function>
```

## B.1 Package Subgraphs

Each package has its own independent dependency graph. These vertices may have *virtual edges* to other packages' graphs—edges that will be formed once combined the referenced graph; these edges are indicated with `preproc:sym-ref/@src`.

Graph operations are usually performed on single packages, but it is occionally necessary to traverse packages to recurisvely resolve dependencies. [graph:dep-lookup#3], page 14 makes this easy:

TODO: Generic graph functions.

`element( preproc:sym-dep )? graph:dep-lookup` (*lookup as xs:sequence**, *graph as element( preproc:sym-deps )*, *symbol as element( preproc:sym )*) [function]

`xmlns:graph="http://www.lovullo.com/tame/graph"`

Retrieve dependenices for *$symbol* on the *$graph*, using the lookup function *$lookup* to resolve external subgraphs. *$lookup* will be used only if the symbol cannot be found in *$graph*, in which case the result of *$lookup* will used used in a recursive call as the new *$graph*.

From a graph perspective, the dependencies are edges on the *$symbol* vertex.

Parameters are organized for partial application.

*Definition:*

```
<function name="graph:dep-lookup"  as="element( preproc:sym-dep )?">
  <param name="lookup" />
  <param name="graph"  as="element( preproc:sym-deps )" />
  <param name="symbol"  as="element( preproc:sym )" />

  <variable name="deps"  as="element( preproc:sym-dep )?"  select="$graph/preproc:sym-

  <sequence select="if ( exists( $deps ) ) then $deps else if ( $lookup ) then graph:d
</function>
```

[graph:dep-lookup#3], page 14 can be used together with the convenience function [graph:make-from-deps#2], page 15 to produce a graph that contains all dependencies for a given symbol list. Used together with [graph:reverse#1], page 12, a reverse dependency graph can be easily created that provides a useful "used by" relationship.

`element( preproc:sym-deps )* graph:make-from-deps` (*lookup as item()+*, *symbols as element( preproc:sym )**) [function]

`xmlns:graph="http://www.lovullo.com/tame/graph"`

Create a dependency graph containing all dependencies of the given symbol list *$symbols*. The graph contains the union of the minimal subset of all package subgraphs—only vertices representing a symbol in *$symbols* or its direct dependencies are included.

This function is *not* recursive; it assumes that the given symbol list *$symbols* is sufficient for whatever operation is being performed.

The lookup function *$lookup* is invoked once per symbol in *$symbols* with the `preproc:sym` to look up. The final result is used to produce a new normalized graph, with any duplicate vertices and edges removed.

*Definition:*

```
<function name="graph:make-from-deps"  as="element( preproc:sym-deps )*">
  <param name="lookup"  as="item()+" />
  <param name="symbols"  as="element( preproc:sym )*" />

  <sequence select="graph:make-from-vertices( for $symbol in $symbols return f:apply(
</function>
```

### B.1.1 Graph Lookups

The provided graph lookups are constructors that use symbols to locate a graph. Using partial application, they are convenient for use in [graph:dep-lookup#3], page 14 to resolve external graphs.

`element( preproc:sym-deps )? graph:lookup-from-doc` (*doc-ext as xs:string, rel-node as node*(), *symbol as element*( *preproc:sym* )) [function]

`xmlns:graph="http://www.lovullo.com/tame/graph"`

Look up a graph on a document indicated by a source symbol *$symbol/@src*. The document will be loaded relative to *$rel-node* with the file extension *$package-ext*.

There are no restrictions on the root node of the document, but the `preproc:sym-deps` node is expected to be a child of the root.

This function does not care if *$symbol* actually resolves to anything in the destination package— such is up to the caller to decide. If the referenced document contains no graph (`preproc:sym-deps`), the empty sequence will be returned. If the referenced document does not exist, the result is implementation-defined.

Customarily, *$doc-ext* is "xmlo" (the compiled object file) and *$rel-node* is the package from which *$symbol* was obtained.

*Definition:*

```
<function name="graph:lookup-from-doc"  as="element( preproc:sym-deps )?">
  <param name="doc-ext"  as="xs:string" />
  <param name="rel-node"  as="node()" />
  <param name="symbol"  as="element( preproc:sym )" />

  <variable name="src"  as="xs:string?"  select="$symbol/@src" />

  <sequence select="if ( $src ) then document( concat( $src, '.', $doc-ext ), $rel-nod
</function>
```

# Appendix C Symbol Table

The *symbol table* holds declarations for each symbol known to a particular package. Symbol tables are represented by `preproc:syms` elements.[1]

A *symbol* is an abstract representation of some object— a calculation, classification, typedef, etc.— containing the source location and useful metadata. *TODO: Document symbol format and metadata.* Symbols are represented by `preproc:sym` elements.

`element( preproc:sym )* symtable:find-duplicates` (*symtable as element( preproc:syms )*) [function]

`xmlns:symtable="http://www.lovullo.com/tame/symtable"`

Produce a list of duplicate symbols in *$symtable*, grouped by `@name`. All duplicates will be returned— that is, if $S_1$ appears before duplicate $S_2$ in the symbol table, both $S_1$ and $S_2$ will be returned.

If two symbols have duplicate `@name`s but the same `@src`, then they are not considered to be duplicates, *unless* another duplicate symbol of the same `@name` is found with a different `@src`, in which case all symbols will be returned. An exception to this rule is made when both symbols lack a `@src`, meaning that they are both defined in the same package. This allows sloppy comparison on concatenated symbol tables before tidying it up.

Externs are ignored, since they represent symbols that need to be satisfied at some point— this will be checked during linking.

Symbols (`preproc:sym` nodes) are returned by reference.

This method name is "find" duplicates rather than "get" to emphasize that processing is performed, which is potentially intensive given a large symbol table *$symtable*.

*Definition:*

```
<function name="symtable:find-duplicates"  as="element( preproc:sym )*">
  <param name="symtable"  as="element( preproc:syms )" />

  <for-each-group select="$symtable/preproc:sym[ not( @extern = 'true' ) ]"  group-by=
    <!-- @src may be omitted to convey a local symbol -->
    <variable name="srcs"  as="xs:string*"  select="distinct-values( for $sym in curre

    <sequence select="if ( count( $srcs ) gt 1 ) then current-group() else if ( ( $src
  </for-each-group>
</function>
```

## C.1 Symbol Format

A *symbol* is nothing more than an abstract representation of an object, represented by a `preproc:sym` node in the symbol table (see Appendix C [Symbol Table], page 17), For some symbol $\sigma$ describing some object $\theta$, the following attributes are recognized:

`name` Unique identifier of $\sigma$. *Required.*

[1] The `preproc` namespace exists for legacy reasons; it will change in the future.

`type` Type of object described by $\sigma$. Multiple symbols of different types may share the same $\theta$. *Required.*

`desc` Human-readable description of $\theta$. *Required.*

`dim` Dimensions of $\theta$ as an integer. See [_symtable:str-to-dim#1], page 18 for supported strings. *Required*

`dtype` Reference to some symbol describing the datatype of $\sigma$, if applicable.

`tex` TeX code used to render $\sigma$ in a math block.

`parent` Reference to a symbol from which $\sigma$ is derived. For example, a `cgen` symbol would have the associated `class` as its parent.

`extern` Whether $\sigma$ is unresolved in the given context.

`missing` If extern, a human-readable message describing $\theta$. This is useful to display to the user on error when $\sigma$ is the result of generated code (e.g. Section A.1 [template expansion], page 6).

`no-deps` When `true`, linker does not attempt to look up dependencies for this symbol. Otherwise, as a safeguard against compilation bugs, the linker will fail if a symbol is missing a dependency list.

`allow-circular`
Permit $\sigma$ to be part of a cycle (see Appendix B [Dependency Graph], page 12). This is desirable for recursive functions, but should otherwise be avoided like the plague.

`keep` Always link $\sigma$, even if disjoint from the program's dependency graph. *This is marked for removal.*[2]

An attribute not marked as required may be omitted. If an implementation wishes to attach additional metadata to a symbol, it *must* use a different namespace.

`xs:integer _symtable:str-to-dim` (*str as xs:string?*) [function]

`xmlns:_symtable="http://www.lovullo.com/tame/symtable/_priv"`

Convert a string dimension representation into an integer.

Standard dimensions are '`scalar`' (0), '`vector`' (1), and '`matrix`' (2). If no value is provided, then '`scalar`' is assumed. All unknown strings will yield a value of '`-1`'.[3]

*Definition:*

```
<function name="_symtable:str-to-dim"  as="xs:integer">
  <param name="str"  as="xs:string?" />
```

[2] "Keeps" cause various tricky and inelegant situations, cause unnecessary coupling of unrelated concerns, and create a performance nightmare for the linker. They were created as a kluge during a near-rewrite of TAME (which introduced the symbol table) to get around the fact that certain systems weren't in place to pull in symbols as dependencies for certain operations; it wasn't intended to stick. They will be removed entirely and will not be replaced with anything— if a symbol is to be linked, it must be part of the dependency graph.

[3] That's not to say that TAME can't support an arbitrary number of dimensions; this syntax just doesn't provide that utility.

```
    <choose>
      <when test="empty( $str ) or ( $str = 'scalar' )">
        <sequence select="0" />
      </when>

      <when test="$str = 'vector'">
        <sequence select="1" />
      </when>

      <when test="$str = 'matrix'">
        <sequence select="2" />
      </when>

      <otherwise>
        <sequence select="-1" />
      </otherwise>
    </choose>
  </function>
```

## C.2 Symbol Types

The *symbol type* describes the context in which a symbol may be used, and the attributes it supports. Generally, both the compiler and linker must be aware of a given symbol type: the compiler uses the symbol type to determine the type of code to generate, and the linker uses that information to determine what section the object code should be linked to. There are exceptions, noted in their individual sections.

TAME aims to reduce the number of symbol types in favor of a generalized system with few primitives— before defining a new symbol type, consider whether the existing can be reused in a novel way. Certain symbols will be consolidated in the future, time permitting.[4]

`rate` Calculation.[5] Calculations yield a single scalar value. If a child `cgen` exists, the yield of the calculation is equivalent to the sum of all its elements.

`gen` Generator.[6] Generators produce a vector element for each iteration of some calculation. Generators always have a parent `rate` symbol.

`class` Classification. Classifications are universal or existential quantifiers most often used as predicates.

---

[4] `class` can be a special case of `rate`. `gen` can acquire the multidimensional capabilities of `cgen` and the latter can be removed. `param` and `lparam` can be merged if the former is defined by a wide "local" scope. `const` is a special case of `rate` or `cgen`. `func` can be eliminated if all current `rate` are considered to be immediate function applications; this would also allow for mocking inputs for testing and debugging. If it's not yet clear, TAME started as a very rigid, narrowly-focused system and began generalizing over time.

[5] The term *rate* (a verb) is a consequence of TAME's origin as an insurance rating system. This symbol type will be renamed to `calc` eventually.

[6] The name is regrettable— it originated from the concept of a generator function, but never turned out that way. `gen` is actually a list comprehension, and will likely be renamed in the future to reflect that.

`cgen` Classification generator.[7] Analogous to `gen`, produces a boolean element for each classification result. Unlike `gen`, the dimension of a given `cgen` is the largest of all of its predicates.

`param` Global parameter. All TAME programs can be viewed as a function operating on a set of global inputs. Their domains are defined by their datatype, represented by the symbol attribute `dtype` (see Section C.2 [Symbol Types], page 19).

`lparam` Local parameter. In contrast to `param`, local parameters are restricted to a defined scope (e.g. function parameters; let expressions). This symbol is not used by the linker.

`lparent` Local parameter parent. This is set for `let` expressions in place of `@parent` since the parent let does not actually generate a symbol.

`const` Global constant. Constant values (of any dimension). These values may be inlined at the compiler's discretion.

`tpl` Template definition. Templates define expandable text in the form of macros (see Section A.1 [Macro Expansion], page 6). This symbol is not used by the linker.

`type` Datatype. A type describes the domain of a symbol. Types do not include dimension information— such is determined by the `dim` symbol attribute (see Section C.1 [Symbol Format], page 17).

`func` Function. TAME supports functions as a means of calculation reuse and abstraction. Unlike calculations, functions can recurse (see `allow-circular` in Section C.1 [Symbol Format], page 17). Functions are not first-class— TAME is not a functional language.

`meta` Arbitrary metadata about the program. These metadata are key-value and are intended to be compiled statically into the program for static reference without having to actually invoke the program.

### C.2.1 Parameter Symbols

Global parameters define all inputs to the program.

`preproc:symtable` *on lv:param* [match]

Produce a `param` symbol with the following attributes:

`name` Name of the parameter as provided by `lv:param/@name`.

`type` The datatype defining the parameter's domain, as provided by `lv:param/@type`.

`dim` Numeric dimension converted from its string representation in `lv:param/@set`.[8]

---

[7] See footnote for `gen`.

[8] The attribute name `lv:param/@set` is unfortunate and simply incorrect terminology with how it is used. It will be changed in the future.

`desc` Description as provided by `lv:param/@desc`.

`tex` TeX symbol used when rendering parameter in an equation.

`keep` Always '`true`' to ensure that the symbol is retained after linking.

```
<template match="lv:param"  mode="preproc:symtable"  priority="5">
  <variable name="dim"  as="xs:integer"  select="_symtable:str-to-dim( @set )" />

  <preproc:sym type="param"  name="{@name}"  dim="{$dim}"  desc="{@desc}"  dtype="{@ty
</template>
```

### C.2.2 Template Symbols

Templates produce a single `tpl` symbol representing the macro itself. At the time of parsing, we do not care about the body of the template— when applied, it will expand into code that will be further processed recursively during another pass to produce the appropriate symbols for that expansion.

`preproc:symtable` *on lv:template* [match]

Produce a `tpl` symbol with the following attributes:

`name` Name of the template as provided by `lv:template/@name`.

`dim` Always '`0`'; templates are processed as macros before compilation.

`desc` Template description as provided by `lv:template/@desc`.

```
<template mode="preproc:symtable"  priority="5"  match="lv:template">
  <preproc:sym type="tpl"  name="{@name}"  dim="0"  desc="{@desc}">
    <if test="@preproc:generated = 'true'">
      <attribute name="local"  select="'true'" />
    </if>

    <sequence select="@preproc:*" />
  </preproc:sym>
</template>
```

### C.2.3 Program Metadata Symbols

A basic key-value system allows for compiling static metadata into the program. These metadata can be referenced externally without having to run the program.

`preproc:symtable` *on lv:meta* [match]

Produce a `meta` symbol for each `lv:meta/lv:prop` with the following attributes:

`name` The metavalue name as provided by `lv:prop/@name`, prefixed with '`:meta:`' to avoid conflicts with other symbols.

`desc` Generic description including `name`.

`keep` Always '`true`'.

The `name` prefix enforces separation between the two compilation stages by preventing conflicts.

```
<template mode="preproc:symtable"  priority="5"  match="lv:meta">
  <for-each select="lv:prop">
    <preproc:sym type="meta"  name=":meta:{@name}"  desc="Metavalue {@name}"  pollute=
  </for-each>
</template>
```

# Appendix D GNU Free Documentation License

Version 1.3, 3 November 2008



0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Concept Index

# Developer Notes Index

## M



## T