Haddock Documentation

Release 1.0

Simon Marlow

Apr 26, 2024

CONTENTS

1 Introduction **3**
- 1.1 Obtaining Haddock . . . 4
- 1.2 License . . . 4
- 1.3 Contributors . . . 4
- 1.4 Acknowledgements . . . 4

2 Invoking Haddock **5**
- 2.1 Using literate or pre-processed source . . . 11
- 2.2 Avoiding recompilation . . . 11

3 Documentation and Markup **13**
- 3.1 Documenting a Top-Level Declaration . . . 13
- 3.2 Documenting Parts of a Declaration . . . 14
- 3.3 The Module Description . . . 16
- 3.4 Controlling the Documentation Structure . . . 18
- 3.5 Hyperlinking and Re-Exported Entities . . . 23
- 3.6 Module Attributes . . . 24
- 3.7 Markup . . . 25

4 Common Errors **33**
- 4.1 `parse error on input -- | xxx` . . . 33
- 4.2 `parse error on input -- $ xxx` . . . 33

5 Haddocks of multiple components **35**

6 Indices and tables **37**

Index **39**

This is Haddock, a tool for automatically generating documentation from annotated Haskell source code.

Contents:

CHAPTER

ONE

INTRODUCTION

This is Haddock, a tool for automatically generating documentation from annotated Haskell source code. Haddock was designed with several goals in mind:

- When documenting APIs, it is desirable to keep the documentation close to the actual interface or implementation of the API, preferably in the same file, to reduce the risk that the two become out of sync. Haddock therefore lets you write the documentation for an entity (function, type, or class) next to the definition of the entity in the source code.
- There is a tremendous amount of useful API documentation that can be extracted from just the bare source code, including types of exported functions, definitions of data types and classes, and so on. Haddock can therefore generate documentation from a set of straight Haskell 98 modules, and the documentation will contain precisely the interface that is available to a programmer using those modules.
- Documentation annotations in the source code should be easy on the eye when editing the source code itself, so as not to obscure the code and to make reading and writing documentation annotations easy. The easier it is to write documentation, the more likely the programmer is to do it. Haddock therefore uses lightweight markup in its annotations, taking several ideas from IDoc. In fact, Haddock can understand IDoc-annotated source code.
- The documentation should not expose any of the structure of the implementation, or to put it another way, the implementer of the API should be free to structure the implementation however he or she wishes, without exposing any of that structure to the consumer. In practical terms, this means that while an API may internally consist of several Haskell modules, we often only want to expose a single module to the user of the interface, where this single module just re-exports the relevant parts of the implementation modules.

  Haddock therefore understands the Haskell module system and can generate documentation which hides not only non-exported entities from the interface, but also the internal module structure of the interface. A documentation annotation can still be placed next to the implementation, and it will be propagated to the external module in the generated documentation.
- Being able to move around the documentation by following hyperlinks is essential. Documentation generated by Haddock is therefore littered with hyperlinks: every type and class name is a link to the corresponding definition, and user-written documentation annotations can contain identifiers which are linked automatically when the documentation is generated.
- We might want documentation in multiple formats - online and printed, for example. Haddock comes with HTML, LaTeX, and Hoogle backends, and it is structured in such a way that adding new backends is straightforward.

1.1 Obtaining Haddock

Haddock is distributed with GHC distributions, and will automatically be provided if you use ghcup, for instance.

Up-to-date sources can also be obtained from our public GitHub repository. The Haddock sources are at `https://github.com/haskell/haddock`.

1.2 License

The following license covers this documentation, and the Haddock source code, except where otherwise indicated.

Copyright (c) 2002-2010, Simon Marlow All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS AS IS AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.3 Contributors

A list of contributors to the project can be seen at `https://github.com/haskell/haddock/graphs/contributors`.

1.4 Acknowledgements

Several documentation systems provided the inspiration for Haddock, most notably:

- IDoc
- HDoc
- Doxygen

and probably several others Ive forgotten.

Thanks to the the members of haskelldoc@haskell.org, haddock@projects.haskell.org and everyone who contributed to the many libraries that Haddock makes use of.

CHAPTER

TWO

INVOKING HADDOCK

Haddock is invoked from the command line, like so:

```
haddock [option ...] file ...
```

Where each `file` is a filename containing a Haskell source module (.hs) or a Literate Haskell source module (.lhs) or just a module name.

All the modules specified on the command line will be processed together. When one module refers to an entity in another module being processed, the documentation will link directly to that entity.

Entities that cannot be found, for example because they are in a module that isnt being processed as part of the current batch, simply wont be hyperlinked in the generated documentation. Haddock will emit warnings listing all the identifiers it couldnt resolve.

The modules should *not* be mutually recursive, as Haddock dont like swimming in circles.

Note that while older version would fail on invalid markup, this is considered a bug in the new versions. If you ever get failed parsing message, please report it.

You must also specify an option for the output format. Currently only the *`--html`* option for HTML, the *`--hoogle`* option for outputting Hoogle data, and the *`--latex`* option are functional.

The packaging tool Cabal has Haddock support, and is often used instead of invoking Haddock directly.

The following options are available:

-B <dir>
: Tell GHC that its lib directory is dir. Can be used to override the default path.

-o <dir>
--odir=<dir>
: Generate files into dir instead of the current directory.

-l <dir>
--lib=<dir>
: Use Haddock auxiliary files (themes, javascript, etc) in dir.

-i <file>
--read-interface=<file>
-i <docpath>,<file>
--read-interface=<docpath>,<file>
-i <docpath>,<srcpath>,<file>
--read-interface=<docpath>,<srcpath>,<file>
-i <docpath>,<srcpath>,<visibility>,<file>
: Read the interface file in file, which must have been produced by running Haddock with the

--dump-interface option. The interface describes a set of modules whose HTML documentation is located in docpath (which may be a relative pathname). The docpath is optional, and defaults to .. The srcpath is optional but has no default value.

This option allows Haddock to produce separate sets of documentation with hyperlinks between them. The docpath is used to direct hyperlinks to point to the right files; so make sure you dont move the HTML files later or these links will break. Using a relative docpath means that a documentation subtree can still be moved around without breaking links.

Similarly to docpath, srcpath is used generate cross-package hyperlinks but within sources rendered with *--hyperlinked-source* option.

If visibility is set to *hidden*, modules from that interface file will not be listed in haddock generated content file.

Multiple *--read-interface* options may be given.

-D <file>
--dump-interface=<file>
Produce an interface file[1] in the file file. An interface file contains information Haddock needs to produce more documentation that refers to the modules currently being processed - see the *--read-interface* option for more details. The interface file is in a binary format; dont try to read it.

--show-interface=<file>
Dumps a binary interface file to stdout in a human readable fashion. Uses json as output format.

--html, -h
Generate documentation in HTML format. Several files will be generated into the current directory (or the specified directory if the *-o* option is given), including the following:

module.html; mini_module.html An HTML page for each module, and a mini page for each used when viewing their synopsis.

index.html The top level page of the documentation: lists the modules available, using indentation to represent the hierarchy if the modules are hierarchical.

doc-index.html; doc-index-X.html The alphabetic index, possibly split into multiple pages if big enough.

some.css; etc... Files needed for the themes used. Specify your themes using the *--theme* option.

haddock-util.js Some JavaScript utilities used to implement some of the dynamic features like collapsible sections.

--mathjax
Specify a custom URL for a mathjax-compatible JS script. By default, this is set to MathJax.

--latex
Generate documentation in LaTeX format. Several files will be generated into the current directory (or the specified directory if the *-o* option is given), including the following:

package.tex The top-level LaTeX source file; to format the documentation into PDF you might run something like this:

```
$ pdflatex package.tex
```

haddock.sty The default style. The file contains definitions for various macros used in the LaTeX sources generated by Haddock; to change the way the formatted output looks, you might want to override these by specifying your own style with the *--latex-style* option.

module.tex The LaTeX documentation for each module.

[1] Haddock interface files are not the same as Haskell interface files, I just couldnt think of a better name.

--latex-style=<style>
This option lets you override the default style used by the LaTeX generated by the *--latex* option. Normally Haddock puts a standard `haddock.sty` in the output directory, and includes the command `\usepackage{haddock}` in the LaTeX source. If this option is given, then `haddock.sty` is not generated, and the command is instead `\usepackage{style}`.

--hoogle
Generate an index file for the Hoogle search engine. One text file will be generated into the current directory (or the specified directory if the *-o* is given). Note that the *--package-name* is required.

Since the output is intended to be parsed by Hoogle, some conventions need to be upheld:

- Every entity should span exactly one line.

```
newtype ReaderT r (m :: * -> *) a :: * -> (* -> *) -> * -> *
```

The one exception to this rule is classes. The body of a class is split up with one class member per line, an opening brace on the line of the header, and a closing brace on a new line after the class.

```
class Foo a where {
    foo :: a -> a -> Baz a;
    type family Baz a;
    type Baz a = [(a, a)];
}
```

- Entites that are exported only indirectly (for instance data constructors visible via a `ReaderT(..)` export) have their names wrapped in square brackets.

```
[ReaderT] :: (r -> m a) -> ReaderT r m a
[runReaderT] :: ReaderT r m a -> r -> m a
```

--hyperlinked-source
Generate hyperlinked source code (as HTML web page). All rendered files will be put into `src/` subfolder of output directory.

Usually, this should be used in combination with *--html* option - generated documentation will then contain references to appropriate code fragments. Previously, this behaviour could be achieved by generating sources using external tool and specifying *--source-base*, *--source-module*, *--source-entity* and related options. Note that these flags are ignored once *--hyperlinked-source* is set.

In order to make cross-package source hyperlinking possible, appropriate source paths have to be set up when providing interface files using *--read-interface* option.

--source-css=<style>
Use custom CSS file for sources rendered by the *--hyperlinked-source* option. If no custom style file is provided, Haddock will use default one.

-S, --docbook
Reserved for future use (output documentation in DocBook XML format).

--base-url=<url>
Base url for static assets (eg. css, javascript, json files etc.). When present, static assets are not copied. This option is useful when creating documentation for multiple packages, it allows to have a single copy of static assets served from the given url.

--source-base=<url>
--source-module=<url>
--source-entity=<url>
--source-entity-line=<url>
Include links to the source files in the generated documentation. Use the *--source-base* option to add a

source code link in the header bar of the contents and index pages. Use the *--source-module* to add a source code link in the header bar of each module page. Use the *--source-entity* option to add a source code link next to the documentation for every value and type in each module. *--source-entity-line* is a flag that gets used for entities that need to link to an exact source location rather than a name, eg. since they were defined inside a Template Haskell splice.

In each case URL is the base URL where the source files can be found. For the per-module and per-entity URLs, the following substitutions are made within the string URL:

- The string `%M` or `%{MODULE}` is replaced by the module name. Note that for the per-entity URLs this is the name of the *exporting* module.
- The string `%N` or `%{NAME}` is replaced by the name of the exported value or type. This is only valid for the *--source-entity* option.
- The string `%K` or `%{KIND}` is replaced by a flag indicating whether the exported name is a value `v` or a type `t`. This is only valid for the *--source-entity* option.
- The string `%L` or `%{LINE}` is replaced by the number of the line where the exported value or type is defined. This is only valid for the *--source-entity* option.
- The string `%%` is replaced by `%`.

If you have html versions of your sources online with anchors for each type and function name, you would say `haddock --source-base=url/ --source-module=url/%M.html --source-entity=url/%M.html#%N`

For the `%{MODULE}` substitution you may want to replace the `.` character in the module names with some other character (some web servers are known to get confused by multiple `.` characters in a file name). To replace it with a character c use `%{MODULE/./c}`.

One example of a tool that can generate syntax-highlighted HTML from your source code, complete with anchors suitable for use from haddock, is hscolour.

-s <url>
--source=<url>
Deprecated aliases for *--source-module*

--comments-base=<url>
--comments-module=<url>
--comments-entity=<url>
documentation. This feature would typically be used in conjunction with a Wiki system.

Use the *--comments-base* option to add a user comments link in the header bar of the contents and index pages. Use the *--comments-module* to add a user comments link in the header bar of each module page. Use the *--comments-entity* option to add a comments link next to the documentation for every value and type in each module.

In each case URL is the base URL where the corresponding comments page can be found. For the per-module and per-entity URLs the same substitutions are made as with the *--source-module* and *--source-entity* options above.

For example, if you want to link the contents page to a wiki page, and every module to subpages, you would say `haddock --comments-base=url --comments-module=url/%M`

If your Wiki system doesnt like the `.` character in Haskell module names, you can replace it with a different character. For example to replace the `.` characters with `_` use `haddock --comments-base=url --comments-module=url/%{MODULE/./_}`. Similarly, you can replace the `/` in a file name (may be useful for entity comments, but probably not).

--theme=<path>
Specify a theme to be used for HTML (*--html*) documentation. If given multiple times then the pages will use

the first theme given by default, and have alternate style sheets for the others. The reader can switch between themes with browsers that support alternate style sheets, or with the Style menu that gets added when the page is loaded. If no themes are specified, then just the default built-in theme (Linuwial) is used.

The path parameter can be one of:

- A *directory*: The base name of the directory becomes the name of the theme. The directory must contain exactly one `some.css` file. Other files, usually image files, will be copied, along with the `some.css` file, into the generated output directory.
- A *CSS file*: The base name of the file becomes the name of the theme.
- The *name* of a built-in theme (Linuwial, Ocean, or Classic).

--built-in-themes
Includes the built-in themes (Linuwial, Ocean, and Classic). Can be combined with *--theme*. Note that order matters: The first specified theme will be the default.

--use-unicode
Enable use of Unicode characters in HTML output.

-c <file>
--css=<file>
Deprecated aliases for *--theme*

-p <file>
--prologue=<file>
Specify a file containing documentation which is placed on the main contents page under the heading Description. The file is parsed as a normal Haddock doc comment (but the comment markers are not required).

-t <title>
--title=<title>
Use title as the page heading for each page in the documentation.This will normally be the name of the library being documented.

The title should be a plain string (no markup please!).

--package-name=<name>
Specify the name of the package being documented.

--package-version=<version>
Specify the version of the package being documented.

-q <mode>
--qual=<mode>
Specify how identifiers are qualified.

mode should be one of

- `none` (default): dont qualify any identifiers
- `full`: always qualify identifiers completely
- `local`: only qualify identifiers that are not part of the module
- `relative`: like local, but strip name of the module from qualifications of identifiers in submodules

Example: If you generate documentation for module A, then the identifiers A.x, A.B.y and C.z are qualified as follows.

- none: x, y, z
- full: A.x, A.B.y, C.z
- local: x, A.B.y, C.z

- relative: x, B.y, C.z

--since-qual=<mode>
Specify how `@since` annotations are qualified.

mode should be one of

- `always` (default): always qualify `@since` annotations with a package name and version
- `only-external`: only qualify `@since` annotations with a package name and version when they do not come from the current package

-?
--help
Display help and exit.

-V
--version
Output version information and exit.

--ghc-version
Output the version of GHC which Haddock expects to find at :option:-B and exit.

--print-ghc-path
Output the path to the GHC (which Haddock computes based on :option:-B) and exit.

--print-ghc-libdir
Output the path to the GHC `lib` directory (which Haddock computes based on :option:-B) and exit.

-v
--verbose
Increase verbosity. Currently this will cause Haddock to emit some extra warnings, in particular about modules which were imported but it had no information about (this is often quite normal; for example when there is no information about the `Prelude`).

--use-contents=<url>
--use-index=<url>
When generating HTML, do not generate an index. Instead, redirect the Contents and/or Index link on each page to URL. This option is intended for use in conjunction with *--gen-contents* and/or *--gen-index* for generating a separate contents and/or index covering multiple libraries.

--gen-contents
--gen-index
Generate an HTML contents and/or index containing entries pulled from all the specified interfaces (interfaces are specified using *-i* or *--read-interface*). This is used to generate a single contents and/or index for multiple sets of Haddock documentation.

--hide <module>
Causes Haddock to behave as if module module has the `hide` attribute. (*Module Attributes*).

--show <module>
Causes Haddock to behave as if module module does not have the `hide` attribute. (*Module Attributes*).

--show-all
Causes Haddock to behave as if no modules have the `hide` attribute. (*Module Attributes*).

--show-extensions <module>
Causes Haddock to behave as if module module has the `show-extensions` attribute. (*Module Attributes*).

--optghc=<option>
Pass option to GHC. Note that there is a double dash there, unlike for GHC.

-w

--no-warnings
Turn off all warnings.

--interface-version
Prints out the version of the binary Haddock interface files that this version of Haddock generates.

--compatible-interface-versions
Prints out space-separated versions of binary Haddock interface files that this version of Haddock is compatible with.

--bypass-interface-version-check
DANGEROUS Causes Haddock to ignore the interface versions of binary Haddock interface files. This can make Haddock crash during deserialization of interface files.

--no-tmp-comp-dir
Do not use a temporary directory for reading and writing compilation output files (`.o`, `.hi`, and stub files). Instead, use the present directory or another directory that you have explicitly told GHC to use via the *`--optghc`* flag.

This flag can be used to avoid recompilation if compilation files already exist. Compilation files are produced when Haddock has to process modules that make use of Template Haskell, in which case Haddock compiles the modules using the GHC API.

--print-missing-docs
Print extra information about any undocumented entities.

--trace-args
Make Haddock print the arguments it receives to standard output. This is useful for examining arguments when invoking through `cabal haddock`, as `cabal` uses temporary response files to pass arguments to Haddock.

2.1 Using literate or pre-processed source

Since Haddock uses GHC internally, both plain and literate Haskell sources are accepted without the need for the user to do anything. To use the C pre-processor, however, the user must pass the `-cpp` option to GHC using *`--optghc`*.

2.2 Avoiding recompilation

With the advent of hi-haddock, Haddock now produces documentation from `.hi` (Haskell interface) files and `.hie` (`.hi` extended) files[2], rather than typechecked module results. This means that as long as the necessary `.hi` and `.hie` files are available (i.e. produced by your build process), recompilation can be avoided during documentation generation.

The first step is to ensure that your build process is producing `.hi` files that contain Haddock docstrings. This requires that you somehow provide the `-fwrite-interface` and `-haddock` flags to GHC. If you intend to generate documentation that includes hyperlinked source files, you should also provide the `-fwrite-ide-info` flag to GHC. You may specify the directory in which GHC should write the `.hi` and `.hie` files by providing the `-hidir=/path/to/hidir` and `-hiedir=/path/to/hiedir` flags to GHC. If you are building your application with `cabal build`, the default location is in `dist-newstyle/build/<arch>-<os>/ghc-<ghc-version>/<component>-0.1.0/build`.

The next step is to ensure that the flags which Haddock passes to GHC will not trigger recompilation. Unfortunately, this is not very easy to do if you are invoking Haddock through `cabal haddock`. Upon `cabal haddock`, Cabal passes a `--optghc="-optP-D__HADDOCK_VERSION__=NNNN"` (where `NNNN` is the Haddock version number)

[2] Note that `.hie` files are only necessary to build documentation which includes hyperlinked source files like this one, while `.hi` files are required for all Haddock documentation flavors.

flag to Haddock, which forwards the `-optP=...` flag to GHC and triggers a recompilation (unless the existing build results were also created by a `cabal haddock`). Additionally, Cabal passes a `--optghc="-stubdir=<temp directory>"` flag to Haddock, which forwards the `-stubdir=<temp directory>` flag to GHC and triggers a recompilation since `-stubdir` adds a global include directory. Moreover, since the `stubdir` that Cabal passes is a temporary directory, a recompilation is triggered even for immediately successive invocations. To avoid recompilations due to these flags, one must manually extract the arguments passed to Haddock by Cabal and remove the `--optghc="-optP-D__HADDOCK_VERSION__=NNNN"` and `--optghc="-stubdir=<temp directory>"` flags. This can be achieved using the *`--trace-args`* flag by invoking `cabal haddock` with `--haddock-option="--trace-args"` and copying the traced arguments to a script which makes an equivalent call to Haddock without the aformentioned flags.

In addition to the above, Cabal passes a temporary directory as `-hidir` to Haddock by default. Obviously, this also triggers a recompilation for every invocation of `cabal haddock`, since it will never find the necessary interface files in that temporary directory. To remedy this, pass a `--optghc="-hidir=/path/to/hidir"` flag to Haddock, where `/path/to/hidir` is the path to the directory in which your build process is writing `.hi` files.

Following the steps above will allow you to take full advantage of hi-haddock and generate Haddock documentation from existing build results without requiring any further compilation.

CHAPTER

THREE

DOCUMENTATION AND MARKUP

Haddock understands special documentation annotations in the Haskell source file and propagates these into the generated documentation. The annotations are purely optional: if there are no annotations, Haddock will just generate documentation that contains the type signatures, data type declarations, and class declarations exported by each of the modules being processed.

3.1 Documenting a Top-Level Declaration

The simplest example of a documentation annotation is for documenting any top-level declaration (function type signature, type declaration, or class declaration). For example, if the source file contains the following type signature:

```
square :: Int -> Int
square x = x * x
```

Then we can document it like this:

```
-- |The 'square' function squares an integer.
square :: Int -> Int
square x = x * x
```

The `-- |` syntax begins a documentation annotation, which applies to the *following* declaration in the source file. Note that the annotation is just a comment in Haskell it will be ignored by the Haskell compiler.

The declaration following a documentation annotation should be one of the following:

- A type signature for a top-level function,
- A definition for a top-level function with no type signature,
- A `data` declaration,
- A `pattern` declaration,
- A `newtype` declaration,
- A `type` declaration
- A `class` declaration,
- An `instance` declaration,
- A `data family` or `type family` declaration, or
- A `data instance` or `type instance` declaration.

If the annotation is followed by a different kind of declaration, it will probably be ignored by Haddock.

Some people like to write their documentation *after* the declaration; this is possible in Haddock too:

```
square :: Int -> Int
-- ^The 'square' function squares an integer.
square x = x * x
```

Since Haddock uses the GHC API internally, it can infer types for top-level functions without type signatures. However, youre encouraged to add explicit type signatures for all top-level functions, to make your source code more readable for your users, and at times to avoid GHC inferring overly general type signatures that are less helpful to your users.

Documentation annotations may span several lines; the annotation continues until the first non-comment line in the source file. For example:

```
-- |The 'square' function squares an integer.
-- It takes one argument, of type 'Int'.
square :: Int -> Int
square x = x * x
```

You can also use Haskells nested-comment style for documentation annotations, which is sometimes more convenient when using multi-line comments:

```
{-|
  The 'square' function squares an integer.
  It takes one argument, of type 'Int'.
-}
square :: Int -> Int
square x = x * x
```

3.2 Documenting Parts of a Declaration

In addition to documenting the whole declaration, in some cases we can also document individual parts of the declaration.

3.2.1 Class Methods

Class methods are documented in the same way as top level type signatures, by using either the `-- |` or `-- ^` annotations:

```
class C a where
   -- | This is the documentation for the 'f' method
   f :: a -> Int
   -- | This is the documentation for the 'g' method
   g :: Int -> a
```

Associated type and data families can also be annotated in this way.

3.2.2 Constructors and Record Fields

Constructors are documented like so:

```
data T a b
  -- | This is the documentation for the 'C1' constructor
  = C1 a b
  -- | This is the documentation for the 'C2' constructor
  | C2 a b
```

or like this:

```
data T a b
  = C1   -- ^ This is the documentation for the 'C1' constructor
      a  -- ^ This is the documentation for the argument of type 'a'
      b  -- ^ This is the documentation for the argument of type 'b'
```

There is one edge case that is handled differently: only one `-- ^` annotation occurring after the constructor and all its arguments is applied to the constructor, not its last argument:

```
data T a b
  = C1 a b  -- ^ This is the documentation for the 'C1' constructor
  | C2 a b  -- ^ This is the documentation for the 'C2' constructor
```

Record fields are documented using one of these styles:

```
data R a b =
  C { -- | This is the documentation for the 'a' field
      a :: a,
      -- | This is the documentation for the 'b' field
      b :: b
    }

data R a b =
  C { a :: a  -- ^ This is the documentation for the 'a' field
    , b :: b  -- ^ This is the documentation for the 'b' field
    }
```

Alternative layout styles are generally accepted by Haddock - for example doc comments can appear before or after the comma in separated lists such as the list of record fields above.

In cases where more than one constructor exports a field with the same name, the documentation attached to the first occurrence of the field will be used, even if a comment is not present.

```
data T a = A { someField :: a -- ^ Doc for someField of A
             }
         | B { someField :: a -- ^ Doc for someField of B
             }
```

In the above example, all occurrences of `someField` in the documentation are going to be documented with `Doc for someField of A`. Note that Haddock versions 2.14.0 and before would join up documentation of each field and render the result. The reason for this seemingly weird behaviour is the fact that `someField` is actually the same (partial) function.

3.2.3 Deriving clauses

Most instances are top-level, so can be documented as in *Documenting a Top-Level Declaration*. The exception to this is instance that are come from a `deriving` clause on a datatype declaration. These can be documented like this:

```
data D a = L a | M
  deriving ( Eq   -- ^ @since 4.5
           , Ord  -- ^ default 'Ord' instance
           )
```

This also scales to the various GHC extensions for deriving:

```
newtype T a = T a
  deriving          Show      -- ^ derivation of 'Show'
  deriving stock   ( Eq        -- ^ stock derivation of 'Eq'
                   , Foldable -- ^ stock derivation of 'Foldable'
                   )
  deriving newtype  Ord       -- ^ newtype derivation of 'Ord'
  deriving anyclass Read      -- ^ unsafe derivation of 'Read'
  deriving         ( Eq1      -- ^ deriving 'Eq1' via 'Identity'
                   , Ord1     -- ^ deriving 'Ord1' via 'Identity'
                   ) via Identity
```

3.2.4 Function Arguments

Individual arguments to a function may be documented like this:

```
f  :: Int      -- ^ The 'Int' argument
   -> Float    -- ^ The 'Float' argument
   -> IO ()    -- ^ The return value
```

Pattern synonyms, GADT-style data constructors, and class methods also support this style of documentation.

3.3 The Module Description

A module itself may be documented with multiple fields that can then be displayed by the backend. In particular, the HTML backend displays all the fields it currently knows about. We first show the most complete module documentation example and then talk about the fields.

```
{-|
Module      : W
Description : Short description
Copyright   : (c) Some Person, 2013
                  Someone Else, 2014
License     : GPL-3
Maintainer  : sample@email.com
Stability   : experimental
Portability : POSIX

Here is a longer description of this module, containing some
commentary with @some markup@.
-}
module W where
...
```

All fields are optional but they must be in order if they do appear. Multi-line fields are accepted but the consecutive lines have to start indented more than their label. If your label is indented one space, as is often the case with the `--` syntax, the consecutive lines have to start at two spaces at the very least. For example, above we saw a multiline `Copyright` field:

```
{-|
...
Copyright   : (c) Some Person, 2013
                  Someone Else, 2014
...
-}
```

That could equivalently be written as:

```
-- | ...
-- Copyright:
--  (c) Some Person, 2013
--  Someone Else, 2014
-- ...
```

or as:

```
-- | ...
-- Copyright: (c) Some Person, 2013
--     Someone Else, 2014
-- ...
```

but not as:

```
-- | ...
-- Copyright: (c) Some Person, 2013
-- Someone Else, 2014
-- ...
```

since the `Someone` needs to be indented more than the `Copyright`.

Whether new lines and other formatting in multiline fields is preserved depends on the field type. For example, new lines in the `Copyright` field are preserved, but new lines in the `Description` field are not; leading whitespace is not preserved in either[1]. Please note that we do not enforce the format for any of the fields and the established formats are just a convention.

3.3.1 Fields of the Module Description

The `Module` field specifies the current module name. Since the module name can be inferred automatically from the source file, it doesnt affect the output of any of the backends. But you might want to include it for any other tools that might be parsing these comments without the help of GHC.

The `Description` field accepts some short text which outlines the general purpose of the module. If youre generating HTML, it will show up next to the module link in the module index.

The `Copyright`, `License`, `Maintainer` and `Stability` fields should be obvious. An alternative spelling for the `License` field is accepted as `Licence` but the output will always prefer `License`.

The `Portability` field has seen varied use by different library authors. Some people put down things like operating system constraints there while others put down which GHC extensions are used in the module. Note that you might want to consider using the `show-extensions` module flag for the latter (see *Module Attributes*).

[1] Technically, whitespace and newlines in the `Description` field are preserved verbatim by the HTML backend, but because most browsers collapse whitespace in HTML, they dont render as such. But other backends may render this whitespace.

Finally, a module may contain a documentation comment before the module header, in which case this comment is interpreted by Haddock as an overall description of the module itself, and placed in a section entitled `Description` in the documentation for the module. All the usual Haddock *Markup* is valid in this comment.

3.4 Controlling the Documentation Structure

Haddock produces interface documentation that lists only the entities actually exported by the module. If there is no export list then all entities defined by the module are exported.

The documentation for a module will include *all* entities exported by that module, even if they were re-exported from another module. The only exception is when Haddock cant see the declaration for the re-exported entity, perhaps because it isnt part of the batch of modules currently being processed.

To Haddock the export list has even more significance than just specifying the entities to be included in the documentation. It also specifies the *order* that entities will be listed in the generated documentation. This leaves the programmer free to implement functions in any order he/she pleases, and indeed in any *module* he/she pleases, but still specify the order that the functions should be documented in the export list. Indeed, many programmers already do this: the export list is often used as a kind of ad-hoc interface documentation, with headings, groups of functions, type signatures and declarations in comments.

In the next section we give examples illustrating most of the structural markup features. After the examples we go into more detail explaining the related markup, namely *Section Headings*, *(Named) Chunks of Documentation*, and *Re-Exporting an Entire Module*.

3.4.1 Documentation Structure Examples

We now give several examples that produce similar results and illustrate most of the structural markup features. The first two examples use an export list, but the third example does not.

The first example, using an export list with *Section Headings* and inline section descriptions:

```
module Image
  ( -- * Image importers
    --
    -- | There is a "smart" importer, 'readImage', that determines
    -- the image format from the file extension, and several
    -- "dumb" format-specific importers that decode the file as
    -- the specified type.
    readImage
  , readPngImage
  , readGifImage
  , ...
    -- * Image exporters
    -- ...
  ) where

import Image.Types ( Image )

-- | Read an image, guessing the format from the file name.
readImage :: FilePath -> IO Image
readImage = ...

-- | Read a GIF.
readGifImage :: FilePath -> IO Image
readGifImage = ...
```

(continues on next page)

(continued from previous page)

```
-- | Read a PNG.
readPngImage :: FilePath -> IO Image
readPngImage = ...

...
```

Note that the order of the entities `readPngImage` and `readGifImage` in the export list is different from the order of the actual declarations farther down; the order in the export list is the order used in the generated docs. Also, the imported `Image` type itself is not re-exported, so it will not be included in the rendered docs (see *Hyperlinking and Re-Exported Entities*).

The second example, using an export list with a section description defined elsewhere (the `$imageImporters`; see *(Named) Chunks of Documentation*):

```
module Image
  ( -- * Image importers
    --
    -- $imageImporters
    readImage
  , readPngImage
  , readGifImage
  , ...
    -- * Image exporters
    -- ...
  ) where

import Image.Types ( Image )

-- $imageImporters
--
-- There is a "smart" importer, 'readImage', that determines the
-- image format from the file extension, and several "dumb"
-- format-specific importers that decode the file as the specified
-- type.

-- | Read an image, guessing the format from the file name.
readImage :: FilePath -> IO Image
readImage = ...

-- | Read a GIF.
readGifImage :: FilePath -> IO Image
readGifImage = ...

-- | Read a PNG.
readPngImage :: FilePath -> IO Image
readPngImage = ...

...
```

This produces the same rendered docs as the first example, but the source code itself is arguably more readable, since the documentation for the group of importer functions is closer to their definitions.

The third example, without an export list:

```
module Image where
```

(continues on next page)

(continued from previous page)

```
import Image.Types ( Image )

-- * Image importers
--
-- $imageImporters
--
-- There is a "smart" importer, 'readImage', that determines the
-- image format from the file extension, and several "dumb"
-- format-specific importers that decode the file as the specified
-- type.

-- | Read an image, guessing the format from the file name.
readImage :: FilePath -> IO Image
readImage = ...

-- | Read a GIF.
readGifImage :: FilePath -> IO Image
readGifImage = ...

-- | Read a PNG.
readPngImage :: FilePath -> IO Image
readPngImage = ...

...

-- * Image exporters
-- ...
```

Note that the section headers (e.g. `-- * Image importers`) now appear in the module body itself, and that the section documentation is still given using *(Named) Chunks of Documentation*. Unlike in the first example when using an export list, the named chunk syntax `$imageImporters` *must* be used for the section documentation; attempting to use the `-- | ...` syntax to document the image importers here will wrongly associate the documentation chunk with the next definition!

3.4.2 Section Headings

You can insert headings and sub-headings in the documentation by including annotations at the appropriate point in the export list, or in the module body directly when not using an export list.

For example:

```
module Foo (
  -- * Classes
  C(..),
  -- * Types
  -- ** A data type
  T,
  -- ** A record
  R,
  -- * Some functions
  f, g
  ) where
```

Headings are introduced with the syntax `-- *`, `-- **` and so on, where the number of `*`s indicates the level of the heading (section, sub-section, sub-sub-section, etc.).

If you use section headings, then Haddock will generate a table of contents at the top of the module documentation for you.

By default, when generating HTML documentation Haddock will create an anchor to each section of the form `#g:n`, where `n` is an integer that might change as you add new section headings. If you want to create stable links, you can add an explicit anchor (see *Anchors*) after the section heading:

```
module Foo (
  -- * Classes #classes#
  C(..)
) where
```

This will create an HTML anchor `#g:classes` to the section.

The alternative style of placing the commas at the beginning of each line is also supported, e.g.:

```
module Foo (
  -- * Classes
    C(..)
  -- * Types
  -- ** A data type
  , T
  -- ** A record
  , R
  -- * Some functions
  , f
  , g
  ) where
```

When not using an export list, you may insert section headers in the module body. Such section headers associate with all entities declared up until the next section header. For example:

```
module Foo where

-- * Classes
class C a where ...

-- * Types
-- ** A data type
data T = ...

-- ** A record
data R = ...

-- * Some functions
f :: ...
f = ...
g :: ...
g = ...
```

3.4.3 Re-Exporting an Entire Module

Haskell allows you to re-export the entire contents of a module (or at least, everything currently in scope that was imported from a given module) by listing it in the export list:

```
module A (
  module B,
  module C
 ) where
```

What will the Haddock-generated documentation for this module look like? Well, it depends on how the modules `B` and `C` are imported. If they are imported wholly and without any `hiding` qualifiers, then the documentation will just contain a cross-reference to the documentation for `B` and `C`.

However, if the modules are not *completely* re-exported, for example:

```
module A (
  module B,
  module C
 ) where

import B hiding (f)
import C (a, b)
```

then Haddock behaves as if the set of entities re-exported from `B` and `C` had been listed explicitly in the export list.

The exception to this rule is when the re-exported module is declared with the `hide` attribute (see *Module Attributes*), in which case the module is never cross-referenced; the contents are always expanded in place in the re-exporting module.

3.4.4 (Named) Chunks of Documentation

It is often desirable to include a chunk of documentation which is not attached to any particular Haskell declaration, for example, when giving summary documentation for a group of related definitions (see *Documentation Structure Examples*). In addition to including such documentation chunks at the top of the file, as part of the *The Module Description*, you can also associate them with *Section Headings*.

There are several ways to associate documentation chunks with section headings, depending on whether you are using an export list or not:

- The documentation can be included in the export list directly, by preceding it with a `-- |`. For example:

  ```
  module Foo (
     -- * A section heading

     -- | Some documentation not attached to a particular Haskell entity
     ...
   ) where
  ```

  In this case the chunk is not named.

- If the documentation is large and placing it inline in the export list might bloat the export list and obscure the structure, then it can be given a name and placed out of line in the body of the module. This is achieved with a special form of documentation annotation `--ă$`, which we call a *named chunk*:

  ```
  module Foo (
     -- * A section heading
  ```

(continues on next page)

(continued from previous page)

```
  -- $doc
  ...
) where

-- $doc
-- Here is a large chunk of documentation which may be referred to by
-- the name $doc.
```

The documentation chunk is given a name of your choice (here `doc`), which is the sequence of alphanumeric characters directly after the `--ă$`, and it may be referred to by the same name in the export list. Note that named chunks must come *after* any imports in the module body.

- If you arent using an export list, then your only choice is to use a named chunk with the `-- $` syntax. For example:

```
module Foo where

-- * A section heading
--
-- $doc
-- Here is a large chunk of documentation which may be referred to by
-- the name $doc.
```

Just like with entity declarations when not using an export list, named chunks of documentation are associated with the preceding section header here, or with the implicit top-level documentation section if there is no preceding section header.

Warning: the form used in the first bullet above, where the chunk is not named, *does not work* when you arent using an export list. For example:

```
module Foo where

-- * A section heading
--
-- | Some documentation not attached to a particular Haskell entity

-- | The fooifier.
foo :: ...
```

will result in `Some documentation not ...` being attached to the *next* entity declaration, here `foo`, in addition to any other documentation that next entity already has!

3.5 Hyperlinking and Re-Exported Entities

When Haddock renders a type in the generated documentation, it hyperlinks all the type constructors and class names in that type to their respective definitions. But for a given type constructor or class there may be several modules re-exporting it, and therefore several modules whose documentation contains the definition of that type or class (possibly including the current module!) so which one do we link to?

Lets look at an example. Suppose we have three modules `A`, `B` and `C` defined as follows:

```
module A (T) where
data T a = C a
```

(continues on next page)

(continued from previous page)

```
module B (f) where
import A
f :: T Int -> Int
f (C i) = i

module C (T, f) where
import A
import B
```

Module `A` exports a datatype `T`. Module `B` imports `A` and exports a function `f` whose type refers to `T`. Also, both `T` and `f` are re-exported from module C.

Haddock takes the view that each entity has a *home* module; that is, the module that the library designer would most like to direct the user to, to find the documentation for that entity. So, Haddock makes all links to an entity point to the home module. The one exception is when the entity is also exported by the current module: Haddock makes a local link if it can.

How is the home module for an entity determined? Haddock uses the following rules:

- If modules A and B both export the entity, and module A imports (directly or indirectly) module B, then B is preferred.
- A module with the `hide` attribute is never chosen as the home.
- A module with the `not-home` attribute is only chosen if there are no other modules to choose.

If multiple modules fit the criteria, then one is chosen at random. If no modules fit the criteria (because the candidates are all hidden), then Haddock will issue a warning for each reference to an entity without a home.

In the example above, module `A` is chosen as the home for `T` because it does not import any other module that exports `T`. The link from `f`s type in module `B` will therefore point to `A.T`. However, `C` also exports `T` and `f`, and the link from `f`s type in `C` will therefore point locally to `C.T`.

3.6 Module Attributes

Certain attributes may be specified for each module which affect the way that Haddock generates documentation for that module. Attributes are specified in a comma-separated list in an `{-# OPTIONS_HADDOCK ... #-}` pragma at the top of the module, either before or after the module description. For example:

```
{-# OPTIONS_HADDOCK hide, prune #-}

-- |Module description
module A where
...
```

The options and module description can be in either order.

The following attributes are currently understood by Haddock:

`hide` Omit this module from the generated documentation, but nevertheless propagate definitions and documentation from within this module to modules that re-export those definitions.

`prune` Omit definitions that have no documentation annotations from the generated documentation.

`not-home` Indicates that the current module should not be considered to be the home module for each entity it exports, unless that entity is not exported from any other module. See *Hyperlinking and Re-Exported Entities* for more details.

show-extensions Indicates that we should render the extensions used in this module in the resulting documentation. This will only render if the output format supports it. If Language is set, it will be shown as well and all the extensions implied by it wont. All enabled extensions will be rendered, including those implied by their more powerful versions.

print-explicit-runtime-reps Print type variables that have kind `RuntimeRep`. By default, these are defaulted to `LiftedRep` so that end users dont have to see the underlying levity polymorphism. This flag is analogous to GHCs `-fprint-explicit-runtime-reps` flag.

3.7 Markup

Haddock understands certain textual cues inside documentation annotations that tell it how to render the documentation. The cues (or markup) have been designed to be simple and mnemonic in ASCII so the programmer doesnt have to deal with heavyweight annotations when editing documentation comments.

3.7.1 Paragraphs

One or more blank lines separates two paragraphs in a documentation comment.

3.7.2 Special Characters

The following characters have special meanings in documentation comments: `\`, `/`, `'`, `` ` ``, `"`, `@`, `<`, `$`, `#`. To insert a literal occurrence of one of these special characters, precede it with a backslash (`\`).

Additionally, the character > has a special meaning at the beginning of a line, and the following characters have special meanings at the beginning of a paragraph: `*`, `-`. These characters can also be escaped using `\`.

Furthermore, the character sequence >>> has a special meaning at the beginning of a line. To escape it, just prefix the characters in the sequence with a backslash.

3.7.3 Character References

Although Haskell source files may contain any character from the Unicode character set, the encoding of these characters as bytes varies between systems. Consequently, only source files restricted to the ASCII character set are portable. Other characters may be specified in character and string literals using Haskell character escapes. To represent such characters in documentation comments, Haddock supports SGML-style numeric character references of the forms `&#D;` and `&#xH;` where D and H are decimal and hexadecimal numbers denoting a code position in Unicode (or ISO 10646). For example, the references `λ`, `λ` and `λ` all represent the lower-case letter lambda.

3.7.4 Code Blocks

Displayed blocks of code are indicated by surrounding a paragraph with `@...@` or by preceding each line of a paragraph with > (we often call these bird tracks). For example:

```
-- | This documentation includes two blocks of code:
--
-- @
--     f x = x + x
-- @
--
-- >  g x = x * 42
```

There is an important difference between the two forms of code block: in the bird-track form, the text to the right of the > is interpreted literally, whereas the `@...@` form interprets markup as normal inside the code block. In particular, `/` is markup for italics, and so e.g. `@x / y / z@` renders as `x` followed by italic `y` with no slashes, followed by `z`.

3.7.5 Examples

Haddock has markup support for examples of interaction with a *read-eval-print loop (REPL)*. An example is introduced with >>> followed by an expression followed by zero or more result lines:

```
-- | Two examples are given below:
--
-- >>> fib 10
-- 55
--
-- >>> putStrLn "foo\nbar"
-- foo
-- bar
```

Result lines that only contain the string `<BLANKLINE>` are rendered as blank lines in the generated documentation.

3.7.6 Properties

Haddock provides markup for properties:

```
-- | Addition is commutative:
--
-- prop> a + b = b + a
```

This allows third-party applications to extract and verify them.

3.7.7 Hyperlinked Identifiers

Referring to a Haskell identifier, whether it be a type, class, constructor, or function, is done by surrounding it with a combination of single quotes and backticks. For example:

```
-- | This module defines the type 'T'.
```

`` `T` `` is also ok. `'T`` and `` `T' `` are accepted but less common.

If there is an entity `T` in scope in the current module, then the documentation will hyperlink the reference in the text to the definition of `T` (if the output format supports hyperlinking, of course; in a printed format it might instead insert a page reference to the definition).

It is also possible to refer to entities that are not in scope in the current module, by giving the full qualified name of the entity:

```
-- | The identifier 'M.T' is not in scope
```

If `M.T` is not otherwise in scope, then Haddock will simply emit a link pointing to the entity `T` exported from module `M` (without checking to see whether either `M` or `M.T` exist).

Since values and types live in different namespaces in Haskell, it is possible for a reference such as `'X'` to be ambiguous. In such a case, Haddock defaults to pointing to the type. The ambiguity can be overcome by explicitly specifying a namespace, by way of a `v` (for value) or `t` (for type) immediately before the link:

```
-- | An implicit reference to  'X', the type constructor
--   An explicit reference to v'X', the data constructor
--   An explicit reference to t'X', the type constructor
data X = X
```

To make life easier for documentation writers, a quoted identifier is only interpreted as such if the quotes surround a lexically valid Haskell identifier. This means, for example, that it normally isnt necessary to escape the single quote when used as an apostrophe:

```
-- | I don't have to escape my apostrophes; great, isn't it?
```

Nothing special is needed to hyperlink identifiers which contain apostrophes themselves: to hyperlink `foo'` one would simply type `'foo''`. Hyperlinking operators works in exactly the same way.

```
-- | A prefix operator @'(++)'@ and an infix identifier @'`elem`'@.
```

3.7.8 Emphasis, Bold and Monospaced Styled Text

Text can be emphasized, made bold (strong) or monospaced (typewriter font) by surrounding it with slashes, double-underscores or at-symbols:

```
-- | This is /emphasized text/, __bold text__ and @monospaced text@.
```

Note that those styled texts must be kept on the same line:

```
-- | Styles /do not work
-- | when continuing on the next line/
```

Other markup is valid inside emphasized, bold and monospaced text.

Frequent special cases:

- To have a forward slash inside of emphasis, just escape it: `/fo\/o/`.
- Theres no need to escape a single underscore if you need it bold: `__This_text_with_underscores_is_bold__`.
- `@'f'ăaăb@` will hyperlink the identifier `f` inside the code fragment.
- `@__FILE__@` will render `FILE` in bold with no underscores, which may not be what you had in mind.

3.7.9 Linking to Modules

Linking to a module is done by surrounding the module name with double quotes:

```
-- | This is a reference to the "Foo" module.
```

A basic check is done on the syntax of the header name to ensure that it is valid before turning it into a link but unlike with identifiers, whether the module is in scope isnt checked and will always be turned into a link.

It is also possible to specify alternate text for the generated link using syntax analogous to that used for URLs:

```
-- | This is a reference to [the main module]("Module.Main").
```

3.7.10 Itemized and Enumerated Lists

A bulleted item is represented by preceding a paragraph with either `*` or `-`. A sequence of bulleted paragraphs is rendered as an itemized list in the generated documentation, e.g.:

```
-- | This is a bulleted list:
--
--     * first item
--
--     * second item
```

An enumerated list is similar, except each paragraph must be preceded by either `(n)` or `n.` where n is any integer. e.g.

```
-- | This is an enumerated list:
--
--     (1) first item
--
--     2. second item
```

Lists of the same type dont have to be separated by a newline:

```
-- | This is an enumerated list:
--
--     (1) first item
--     2. second item
--
-- This is a bulleted list:
--
--     * first item
--     * second item
```

You can have more than one line of content in a list element:

```
-- |
-- * first item
-- and more content for the first item
-- * second item
-- and more content for the second item
```

You can even nest whole paragraphs inside of list elements. The rules are 4 spaces for each indentation level. Youre required to use a newline before such nested paragraphs:

```
{-|
* Beginning of list
This belongs to the list above!

    > nested
    > bird
    > tracks

    * Next list
    More of the indented list.

        * Deeper

            @
```

(continues on next page)

(continued from previous page)

```
            even code blocks work
            @

            * Deeper

                    1. Even deeper!
                    2. No newline separation even in indented lists.
-}
```

The indentation of the first list item is honoured. That is, in the following example the items are on the same level. Before Haddock 2.16.1, the second item would have been nested under the first item which was unexpected.

```
{-|
    * foo

    * bar
-}
```

3.7.11 Definition Lists

Definition lists are written as follows:

```
-- | This is a definition list:
--
--   [@foo@]: The description of @foo@.
--
--   [@bar@]: The description of @bar@.
```

To produce output something like this:

`foo` The description of `foo`.

`bar` The description of `bar`.

Each paragraph should be preceded by the definition term enclosed in square brackets and followed by a colon. Other markup operators may be used freely within the definition term. You can escape `]` with a backslash as usual.

Same rules about nesting and no newline separation as for bulleted and numbered lists apply.

3.7.12 URLs

A URL can be included in a documentation comment by surrounding it in angle brackets, for example:

```
<http://example.com>
```

If the output format supports it, the URL will be turned into a hyperlink when rendered.

If Haddock sees something that looks like a URL (such as something starting with `http://` or `ssh://`) where the URL markup is valid, it will automatically make it a hyperlink.

3.7.13 Links

Haddock supports Markdown syntax for inline links. A link consists of a link text and a URL. The link text is enclosed in square brackets and followed by the URL enclosed in regular parentheses, for example:

```
[some link](http://example.com)
```

The link text is used as a description for the URL if the output format supports it.

3.7.14 Images

Haddock supports Markdown syntax for inline images. This resembles the syntax for links, but starts with an exclamation mark. An example looks like this:

```
![image description](pathtoimage.png)
```

If the output format supports it, the image will be rendered inside the documentation. The image description is used as replacement text and/or an image title.

3.7.15 Mathematics / LaTeX

Haddock supports LaTeX syntax for rendering mathematical notation. The delimiters are `\[...\]` for displayed mathematics and `\(...\)` for in-line mathematics. An example looks like this:

```
\[
f(a) = \frac{1}{2\pi i}\oint_\gamma \frac{f(z)}{z-a}\,\mathrm{d}z
\]
```

If the output format supports it, the mathematics will be rendered inside the documentation. For example, the HTML backend will display the mathematics via MathJax.

3.7.16 Grid Tables

Inspired by reSTs grid tables, Haddock supports a complete table representation via grid-like ASCII art. Grid tables are described with a visual grid made up of the characters -, =, |, and +. The hyphen (-) is used for horizontal lines (row separators). The equals sign (=) may be used to separate optional header rows from the table body. The vertical bar (|) is used for vertical lines (column separators). The plus sign (+) is used for intersections of horizontal and vertical lines.

```
-- | This is a grid table:
--
-- +------------------------+------------+----------+----------+
-- | Header row, column 1   | Header 2   | Header 3 | Header 4 |
-- | (header rows optional) |            |          |          |
-- +========================+============+==========+==========+
-- | body row 1, column 1   | column 2   | column 3 | column 4 |
-- +------------------------+------------+----------+----------+
-- | body row 2             | Cells may span columns.          |
-- +------------------------+------------+---------------------+
-- | body row 3             | Cells may  | \[                  |
-- +------------------------+ span rows. | f(n) = \sum_{i=1}   |
-- | body row 4             |            | \]                  |
-- +------------------------+------------+---------------------+
```

3.7.17 Anchors

Sometimes it is useful to be able to link to a point in the documentation which doesnt correspond to a particular entity. For that purpose, we allow *anchors* to be included in a documentation comment. The syntax is `#label#`, where label is the name of the anchor. An anchor is invisible in the generated documentation.

To link to an anchor from elsewhere, use the syntax `"module#label"` where module is the module name containing the anchor, and label is the anchor label. The module does not have to be local, it can be imported via an interface. Please note that in Haddock versions 2.13.x and earlier, the syntax was `"module\#label"`. It is considered deprecated and will be removed in the future.

3.7.18 Headings

Headings inside of comment documentation are possible by preceding them with a number of =s. From 1 to 6 are accepted. Extra =s will be treated as belonging to the text of the heading. Note that its up to the output format to decide how to render the different levels.

```
-- |
-- = Heading level 1 with some /emphasis/
-- Something underneath the heading.
--
-- == /Subheading/
-- More content.
--
-- === Subsubheading
-- Even more content.
```

Note that while headings have to start on a new paragraph, we allow paragraph-level content to follow these immediately.

```
-- |
-- = Heading level 1 with some __bold__
-- Something underneath the heading.
--
-- == /Subheading/
-- More content.
--
-- === Subsubheading
-- >>> examples are only allowed at the start of paragraphs
```

As of 2.15.1, theres experimental (read: subject to change or get removed) support for collapsible headers: simply wrap your existing header title in underscores, as per bold syntax. The collapsible section will stretch until the end of the comment or until a header of equal or smaller number of =s.

```
-- |
-- === __Examples:__
-- >>> Some very long list of examples
--
-- ==== This still falls under the collapse
-- Some specialised examples
--
-- === This is does not go into the collapsable section.
-- More content.
```

3.7.19 Metadata

Since Haddock 2.16.0, some support for embedding metadata in the comments has started to appear. The use of such data aims to standardise various community conventions in how such information is conveyed and to provide uniform rendering.

Since

`@since` annotation can be used to convey information about when the function was introduced or when it has changed in a way significant to the user. `@since` is a paragraph-level element. While multiple such annotations are not an error, only the one to appear in the comment last will be used. `@since` has to be followed with a version number, no further description is currently allowed. The meaning of this feature is subject to change in the future per user feedback.

```
-- |
-- Some comment
--
-- @since 1.2.3
```

Additionally, as of GHC 9.10 `@since` in docstrings following export list items:

```
module Mod
    ( wombat -- ^ @since 2.0
    ) where ...
```

This has the function of superceding the `@since` annotation given at the definition site of `wombat` and can be used to record changes in the module exports, particularly when a declaration is re-exported from another module.

CHAPTER

FOUR

COMMON ERRORS

4.1 parse error on input -- | xxx

This is probably caused by the `-- | xxx` comment not following a declaration. I.e. use `-- xxx` instead. See *Documenting a Top-Level Declaration*.

4.2 parse error on input -- $ xxx

Youve probably commented out code like:

```
f x
  $ xxx
```

`-- $` is a special syntax for named chunks, see *(Named) Chunks of Documentation*. You can fix this by escaping the `$`:

```
-- \$ xxx
```

CHAPTER

FIVE

HADDOCKS OF MULTIPLE COMPONENTS

Haddock supports building documentation of multiple components. First, one needs to build haddocks of all components which can be done with:

```
cabal haddock --haddock-html \
              --haddock-quickjump \
              --haddock-option="--use-index=../doc-index.html" \
              --haddock-option="--use-contents=../index.html" \
              --haddock-option="--base-url=.." \
              all
```

The new `--base-url` option will allow to access the static files from the main directory (in this example its the relative `./..` directory). It will also prevent `haddock` from copying its static files to each of the documentation folders, were only need a single copy of them where the `--base-url` option points to.

The second step requires to copy all the haddocks to a common directory, lets say `./docs`, this will depend on your project and it might look like:

```
cp -r dist-newstyle/build/x86_64-linux/ghc-9.0.1/package-a-0.1.0.0/doc/html/package-a/
↪ docs
cp -r dist-newstyle/build/x86_64-linux/ghc-9.0.1/package-b-0.1.0.0/doc/html/package-b/
↪ docs
```

Note that you can also include documentation of other packages in this way, e.g. `base`, but you need to know where it is hidden on your hard-drive.

To build html and js (`quickjump`) indexes one can now invoke `haddock` with:

```
haddock \
  -o docs \
  --quickjump --gen-index --gen-contents \
  --read-interface=package-a,docs/package-a/package-a.haddock \
  --read-interface=package-b,docs/package-b/package-b.haddock
```

Note: the `PATH` in `--read-interface=PATH,...` must be a relative url of a package it points to (relative to the `docs` directory).

Theres an example project which shows how to do that posted here, which haddocks are served on github-pages.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

-?
    command line option, 10
-B <dir>
    command line option, 5
-D <file>
    command line option, 6
-S
    command line option, 7
-V
    command line option, 10
--base-url=<url>
    command line option, 7
--built-in-themes
    command line option, 9
--bypass-interface-version-check
    command line option, 11
--comments-base=<url>
    command line option, 8
--comments-entity=<url>
    command line option, 8
--comments-module=<url>
    command line option, 8
--compatible-interface-versions
    command line option, 11
--css=<file>
    command line option, 9
--docbook
    command line option, 7
--dump-interface=<file>
    command line option, 6
--gen-contents
    command line option, 10
--gen-index
    command line option, 10
--ghc-version
    command line option, 10
--help
    command line option, 10
--hide <module>
    command line option, 10
--hoogle
    command line option, 7
--html
    command line option, 6
--hyperlinked-source
    command line option, 7
--interface-version
    command line option, 11
--latex
    command line option, 6
--latex-style=<style>
    command line option, 6
--lib=<dir>
    command line option, 5
--mathjax
    command line option, 6
--no-tmp-comp-dir
    command line option, 11
--no-warnings
    command line option, 10
--odir=<dir>
    command line option, 5
--optghc=<option>
    command line option, 10
--package-name=<name>
    command line option, 9
--package-version=<version>
    command line option, 9
--print-ghc-libdir
    command line option, 10
--print-ghc-path
    command line option, 10
--print-missing-docs
    command line option, 11
--prologue=<file>
    command line option, 9
--qual=<mode>
    command line option, 9
--read-interface=<file>
    command line option, 5
--show <module>
    command line option, 10
--show-all

command line option, 10
--show-extensions <module>
command line option, 10
--show-interface=<file>
command line option, 6
--since-qual=<mode>
command line option, 10
--source=<url>
command line option, 8
--source-base=<url>
command line option, 7
--source-css=<style>
command line option, 7
--source-entity=<url>
command line option, 7
--source-entity-line=<url>
command line option, 7
--source-module=<url>
command line option, 7
--theme=<path>
command line option, 8
--title=<title>
command line option, 9
--trace-args
command line option, 11
--use-contents=<url>
command line option, 10
--use-index=<url>
command line option, 10
--use-unicode
command line option, 9
--verbose
command line option, 10
--version
command line option, 10
-c <file>
command line option, 9
-h
command line option, 6
-i <file>
command line option, 5
-l <dir>
command line option, 5
-o <dir>
command line option, 5
-p <file>
command line option, 9
-q <mode>
command line option, 9
-s <url>
command line option, 8
-t <title>
command line option, 9
-v
command line option, 10
-w
command line option, 10

C

command line option
-?, 10
-B <dir>, 5
-D <file>, 6
-S, 7
-V, 10
--base-url=<url>, 7
--built-in-themes, 9
--bypass-interface-version-check, 11
--comments-base=<url>, 8
--comments-entity=<url>, 8
--comments-module=<url>, 8
--compatible-interface-versions, 11
--css=<file>, 9
--docbook, 7
--dump-interface=<file>, 6
--gen-contents, 10
--gen-index, 10
--ghc-version, 10
--help, 10
--hide <module>, 10
--hoogle, 7
--html, 6
--hyperlinked-source, 7
--interface-version, 11
--latex, 6
--latex-style=<style>, 6
--lib=<dir>, 5
--mathjax, 6
--no-tmp-comp-dir, 11
--no-warnings, 10
--odir=<dir>, 5
--optghc=<option>, 10
--package-name=<name>, 9
--package-version=<version>, 9
--print-ghc-libdir, 10
--print-ghc-path, 10
--print-missing-docs, 11
--prologue=<file>, 9
--qual=<mode>, 9
--read-interface=<file>, 5
--show <module>, 10
--show-all, 10
--show-extensions <module>, 10
--show-interface=<file>, 6
--since-qual=<mode>, 10
--source=<url>, 8
--source-base=<url>, 7
--source-css=<style>, 7

--source-entity=<url>, 7
--source-entity-line=<url>, 7
--source-module=<url>, 7
--theme=<path>, 8
--title=<title>, 9
--trace-args, 11
--use-contents=<url>, 10
--use-index=<url>, 10
--use-unicode, 9
--verbose, 10
--version, 10
-c <file>, 9
-h, 6
-i <file>, 5
-l <dir>, 5
-o <dir>, 5
-p <file>, 9
-q <mode>, 9
-s <url>, 8
-t <title>, 9
-v, 10
-w, 10