Haddock User Guide

COLLABORATORS			
	TITLE : Haddock User Guide		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Simon Marlow and David Waern	2004-08-02	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Abstract

This document describes Haddock version 2.11.0, a Haskell documentation tool.

Chapter 1

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 obsure 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 back-ends is straightforward.

1.1 Obtaining Haddock

Distributions (source & binary) of Haddock can be obtained from its web site.

Up-to-date sources can also be obtained from our public darcs repository. The Haddock sources are at `http://code.haskell.org/haddock`. See darcs.net for more information on the darcs version control utility.

1.2 License

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

1.3 Contributors

Haddock was originally written by Simon Marlow. Since it is an open source project, many people have contributed to its development over the years. Below is a list of contributors in alphabetical order that we hope is somewhat complete. If you think you are missing from this list, please contact us.

- Ashley Yakeley
- Benjamin Franksen
- Brett Letner
- Clemens Fruhwirth
- Conal Elliott
- David Waern
- Duncan Coutts
- George Pollard
- George Russel
- Hal Daume
- Ian Lynagh
- Isaac Dupree
- Joachim Breitner
- Krasimir Angelov
- Lennart Augustsson
- Luke Plant

- Malcolm Wallace
- Mark Lentczner
- Mark Shields
- Neil Mitchell
- Mike Thomas
- Manuel Chakravarty
- Oliver Brown
- Roman Cheplyaka
- Ross Paterson
- Simon Hengel
- Simon Marlow
- Simon Peyton-Jones
- Sigbjorn Finne
- Stefan O'Rear
- Sven Panne
- Thomas Schilling
- Wolfgang Jeltsch
- Yitzchak Gale

1.4 Acknowledgements

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

- IDoc
- HDoc
- Doxygen

and probably several others I've 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 2

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 isn't being processed as part of the current batch, simply won't be hyperlinked in the generated documentation. Haddock will emit warnings listing all the indentifiers it couldn't resolve.

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

You must also specify an option for the output format. Currently only the `-h` option for HTML and the `--hoogle` option for outputting Hoogle data 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 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 *path,file* , --read-interface=*path,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 *path* (which may be a relative pathname). The *path* is optional, and defaults to '.'.

This option allows Haddock to produce separate sets of documentation with hyperlinks between them. The *path* is used to direct hyperlinks to point to the right files; so make sure you don't move the HTML files later or these links will break. Using a relative *path* means that a documentation subtree can still be moved around without breaking links.

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; don't try to read it.

-h , --html 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:

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

module.html, mini_module.html An HTML page for each *module*, and a "mini" page for each used when viewing in frames.

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.

frames.html The top level document when viewing in frames.

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 collapsable sections, and switching to frames view.

--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*.

--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}`.

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

--source-base=*URL*, --source-module=*URL*, --source-entity=*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.

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 `%F` or `%{FILE}` is replaced by the original source file name. Note that for the per-entity URLs this is the name of the *defining* 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 `%`.

For example, if your sources are online under some directory, you would say `haddock --source-base=url/ --source-module=url/%F`

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}`.

Similarly, for the `%{FILE}` substitution you may want to replace the '/' character in the file names with some other character (especially for links to colourised entity source code with a shared css file). To replace it with a character *c* use `%{FILE///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* Include links to pages where readers may comment on the 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 doesn't 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 ("Ocean") 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 ("Ocean" or "Classic").

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

-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!).

-q *mode*, --qual=*mode* Specify how identifiers are qualified.

mode should be one of

- none (default): don't 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

-? , --help Display help and exit.

-V , --version Output version information 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 conjuction 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.

--ignore-all-exports Causes Haddock to behaves as if every module has the `ignore-exports` attribute (Section 3.7). This might be useful for generating implementation documentation rather than interface documentation, for example.

--hide *module* Causes Haddock to behaves as if module *module* has the `hide` atribute. (Section 3.7).

--optghc=*option* Pass *option* to GHC.

-w , --no-warnings Turn off all warnings.

--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.

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 the `-cpp` option to GHC using `--optghc`.

Chapter 3

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 `data` declaration,
- A `newtype` declaration,
- A `type` declaration, or
- A `class` 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
```

Note that Haddock doesn't contain a Haskell type system — if you don't write the type signature for a function, then Haddock can't tell what its type is and it won't be included in the documentation.

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 Haskell's 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
```

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 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.

3.2.3 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
```

3.3 The module description

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. For example:

```
-- | This is the description for module "Foo"
module Foo where
...
```

3.4 Controlling the documentation structure

Haddock produces interface documentation that lists only the entities actually exported by the module. The documentation for a module will include *all* entities exported by that module, even if they were re-exported by another module. The only exception is when Haddock can't see the declaration for the re-exported entity, perhaps because it isn't part of the batch of modules currently being processed.

However, 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.

You can insert headings and sub-headings in the documentation by including annotations at the appropriate point in the 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.

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

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

3.4.1 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[1].

The exception to this rule is when the re-exported module is declared with the `hide` attribute (Section 3.7), in which case the module is never cross-referenced; the contents are always expanded in place in the re-exporting module.

3.4.2 Omitting the export list

If there is no export list in the module, how does Haddock generate documentation? Well, when the export list is omitted, e.g.:

```
module Foo where
```

this is equivalent to an export list which mentions every entity defined at the top level in this module, and Haddock treats it in the same way. Furthermore, the generated documentation will retain the order in which entities are defined in the module. In this special case the module body may also include section headings (normally they would be ignored by Haddock).

[1] NOTE: this is not fully implemented at the time of writing (version 0.2). At the moment, Haddock always inserts a cross-reference.

3.5 Named chunks of documentation

Occasionally it is desirable to include a chunk of documentation which is not attached to any particular Haskell declaration. There are two ways to do this:

- The documentation can be included in the export list directly, e.g.:

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

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

- 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 '`-- $`':

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

   -- $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, which is the sequence of alphanumeric characters directly after the '`-- $`', and it may be referred to by the same name in the export list.

3.6 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?

Let's 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

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` atribute 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`.

3.7 Module Attributes

Certain attributes may be specified for each module which affects 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, ignore-exports #-}

-- |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.

`ignore-exports` Ignore the export list. Generate documentation as if the module had no export list - i.e. all the top-level declarations are exported, and section headings may be given in the body of the module.

`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 Section 3.6 for more details.

3.8 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 that the programmer doesn't have to deal with heavyweight annotations when editing documentation comments.

3.8.1 Paragraphs

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

3.8.2 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.

3.8.3 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, so that 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.

3.8.4 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.

3.8.5 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 bellow:
--
-- >>> fib 10
-- 55
--
-- >>> putStrLn "foo\nbar"
-- foo
-- bar
```

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

3.8.6 Hyperlinked Identifiers

Referring to a Haskell identifier, whether it be a type, class, constructor, or function, is done by surrounding it with single quotes:

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

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).

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 isn't necessary to escape the single quote when used as an apostrophe:

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

For compatibility with other systems, the following alternative form of markup is accepted[2]: `` `T' ``.

3.8.7 Emphasis and Monospaced text

Emphasis may be added by surrounding text with `/.../`.

Monospaced (or typewriter) text is indicated by surrounding it with `@...@`. Other markup is valid inside a monospaced span: for example `@'f' a b@` will hyperlink the identifier `f` inside the code fragment.

3.8.8 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.
```

3.8.9 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, eg.:

```
-- | 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
```

3.8.10 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:

[2] We chose not to use this as the primary markup for identifiers because strictly speaking the ` character should not be used as a left quote, it is a grave accent.

`foo` The description of `foo`.

`bar` The description of `bar`.

Each paragraph should be preceded by the "definition term" enclosed in square brackets. The square bracket characters have no special meaning outside the beginning of a definition paragraph. That is, if a paragraph begins with a `[` character, then it is assumed to be a definition paragraph, and the next `]` character found will close the definition term. Other markup operators may be used freely within the definition term.

3.8.11 URLs

A URL can be included in a documentation comment by surrounding it in angle brackets: `<...>`. If the output format supports it, the URL will be turned into a hyperlink when rendered.

3.8.12 Anchors

Sometimes it is useful to be able to link to a point in the documentation which doesn't 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.

Chapter 4

Index