Contents

1 Common.ATermUtils **5**

2 Common.BibTypes **7**

3 Common.HtmlTypes **11**

4 Common.TreeInstances **15**

5 Main **17**

6 ParseBib.Parser **19**

7 ParseBib.ParserUtils **21**

8 ParseBib.Tool **23**

Chapter 1

Common.ATermUtils

```
module Common.ATermUtils (
    aTerm2String,  aTerm2BibTex,  html2Aterm
  ) where
```

This module contains some generally useful functions for manipulating ATerms

`aTerm2String :: Component ATerm String`

Render a given ATerm as a string. Used by all the tools that need to output ATerms to the terminal.

`aTerm2BibTex :: Component ATerm BibTex`

Parse an ATerm into a BibTex data structure.

`html2Aterm :: Component Html ATerm`

This function takes an abstract HTML tree and converts is into an ATerm

Chapter 2

Common.BibTypes

```
module Common.BibTypes (
    BibTex(BibTex),  Entry(Entry, entryType, reference, fields),  Field(Field),
    getKey,  getValue,  maybegetKey,  maybegetValue,  EntryType,  Reference,
    BibTexAlgebra,  foldBibTex,  lookupField,  compareF
  ) where
```

The decision has been made to represent a BibTeX file as a list of entries, along with possibly some preamble.

`data BibTex`

`=  BibTex [String] [Entry]`

a bibtex file is just a list of entries, with a list of preamble-strings

```
instance Show BibTex
instance Tree BibTex
```

`data Entry`

`=` `Entry`

`entryType ::  EntryType` book, thesis, etc. `reference ::  Reference` the name `fields :: [Field]`

a list of key/value pairs

a bibtex entry has a type, a reference (it's name), and a list of fields

```
instance Eq Entry
```

We implement equality on entries. When their names are the same, we consider them equal.

```
instance Show Entry
instance Tree Entry
```

```
data Field
```

```
=  Field String String
```

a field is an attibute/value pair

```
instance Eq Field
```

Fields are considered equal when their keys are the same.

```
instance Ord Field
instance Show Field
instance Tree Field
```

```
getKey :: Field -> String
```

returns the key part, given a Field

```
getValue :: Field -> String
```

returns the value part, given a Field

```
maybegetKey :: Maybe Field -> Maybe String
```

sometimes we want to get the key from a Maybe Field.

```
maybegetValue :: Maybe Field -> Maybe String
```

...and sometimes we want to get the value from a Maybe Field.

```
type EntryType = String
```

the entry type is characterised by a string, for example `book`

```
type Reference = String
```

the reference is also just a string, such as `pierce02`

```
type BibTexAlgebra bibtex preamble entry = ([preamble]
                                            -> [entry] -> bibtex, String -> preamble, EntryType
                                                                                       -> Reference
                                                                                         -> [Field]
                                                                                           -> entry)
```

the type of the bibtex algebra. Used to fold over a bibtex library, like when we want to convert a BibTeX structure into HTML.

This seems the most natural way to define possible conversions from BibTex to other (possibly tree-like) formats, such as Html later on.

```
foldBibTex :: BibTexAlgebra bibtex preamble entry
           -> BibTex -> bibtex
```

How to fold over a BibTeX tree. Used when converting to Html in Bib2HTML.Tool.

```
lookupField :: String -> [Field] -> Maybe Field
```

Given a key, find the corresponding Field in a list of Fields. If it can't be found, Nothing is returned.

```
compareF :: Field -> Field -> Ordering
```

Our implementation of ordering on Fields. This is how we make sure that author, then title, then the other fields, and finally year, are displayed, regardless of how they are placed in the .bib file. This implementation allows us to simply run `sort` on a list of Fields.

Chapter 3

Common.HtmlTypes

```
module Common.HtmlTypes (
    Html(Html),  Head(Head),  Body,  BlockElem(A, Hr, Table, P),  Tr(Tr),  Td,
    Title,  HtmlAlgebra,  foldHtml
  ) where
```

This module contains our representation of an HTML document. Our simplistic version of HTML only knows anchors, horizontal rules, tables and paragraphs, but this is enough to display a BibTeX database.

```
data Html
```

```
     =  Html Head Body
```

an HTML document has a head and a body

```
instance Show Html
instance Tree Html
```

```
data Head
```

```
     =  Head Title
```

the head only contains the title.

```
instance Show Head
instance Tree Head
```

```
type Body = [BlockElem]
```

the body of an HTML document is a list of block elements (table, anchor, etc.)

```
data BlockElem
```

```
=   A [Field] Reference
|   Hr
|   Table [Field] [Tr]
|   P [Field] String
```

a block element can be (as stated) anchor, rule, table or a paragraph (a paragraph is also used to represent a plain string. In this case it's attribute list is empty, and when rendering, the p-tag is ommitted.

```
instance Show BlockElem
instance Tree BlockElem
```

```
data Tr
```

```
=   Tr [Field] [Td]
```

a table row. It has attributes and a list of cells

```
instance Show Tr
instance Tree Tr
```

```
type Td = BlockElem
```

a table cell is a block element

```
type Title = String
```

the title of a document is simply a string.

```
type HtmlAlgebra html head body block tr = (head
                                            -> body -> html, (Title -> head, [block]
                                                                             -> body), ([Field]
                                                                                        -> Refer
                                                                                           -> bl
```

And once again we define an algebra for folding over an HTML document. This will prove useful when we want to pretty-print the html (or render it in any form), since the above type is only an abstract representation of an HTML document.

Since Html is tree-like, it seems logical to use a fold to convert it to some other (tree-like) format, in our case, this is Doc, the CCO pretty-print data structure. The actual fold for this is defined in PrettyPrintHTML.Tool.

```
foldHtml :: HtmlAlgebra html head body block tr -> Html -> html
```

The function which folds over an HTML tree, given an algebra as defined above.

Chapter 4

Common.TreeInstances

```
module Common.TreeInstances (
    bibfromTree,  bibtoTree,  entryfromTree,  entrytoTree,  fieldfromTree,
    fieldtoTree,  htmlfromTree,  htmltoTree,  headfromTree,  headtoTree,
    blockitemfromTree,  blockitemtoTree,  trfromTree,  trtoTree
  ) where
```

This module contains instances of Tree for various data structures. These are used for converting between ATerm and the given data structure. These functions aren't very complicated, they just allow `flattening` a datastructure into a portable format, and converting it back again.

`bibfromTree :: BibTex -> ATerm`

Converts from BibTex to ATerm

`bibtoTree :: ATerm -> Feedback BibTex`

Converts an ATerm back into a BibTex tree

`entryfromTree :: Entry -> ATerm`

Converts from bibtex Entry to ATerm

`entrytoTree :: ATerm -> Feedback Entry`

Converts an ATerm back into a BibTex entry `fieldfromTree :: Field -> ATerm`

Converts from Field (key/value pair) to ATerm

`fieldtoTree :: ATerm -> Feedback Field`

Converts an ATerm back into a key/value pair

`htmlfromTree :: Html -> ATerm`

Converts from HTML document to ATerm

`htmltoTree :: ATerm -> Feedback Html`

Converts an ATerm back into an HTML document

`headfromTree :: Head -> ATerm`

Converts from HTML head element to ATerm

`headtoTree :: ATerm -> Feedback Head`

Converts an ATerm back into an html head element

`blockitemfromTree :: BlockElem -> ATerm`

Converts from HTML entity to ATerm

`blockitemtoTree :: ATerm -> Feedback BlockElem`

Converts an ATerm back into an html element

`trfromTree :: Tr -> ATerm`

Converts from HTML horizontal rule to ATerm

`trtoTree :: ATerm -> Feedback Tr`

Converts an ATerm back into a horizontal rule (html element)

Chapter 5

Main

```
module Main (

  ) where
```

Chapter 6

ParseBib.Parser

```
module ParseBib.Parser (
    parseBib, parseBibEntry, pType, pReference, pBibEntryBody, pBibKey,
    parsePreamble, pKeyValue
  ) where
```

This module implements the parse-bib tool. It uses the UU-Parsinglib parser combinators to allow for a forgiving, self-correcting Bib-parser. It is also quite readable, thanks to the advanced library used.

`parseBib :: Parser BibTex`

> the top-level parser. Describes the grammar of a BibTex file, namely 0 or more preamble keywords, and 1 or more bibtex entries. the definitions here closely resemble the grammar, so it's easy to see what a file should look like, by inspecting the parser functions.

`parseBibEntry :: Parser Entry`

> Grammar for a bib entry. An entry has a type, a name, and a body.

`pType :: Parser String`

> A type of a bib entry is just a string of a-z, preceded by the '@' symbol. Don't return the at-symbol, though.

`pReference :: Parser String`

Parse the name of the bib entry. Cheat by consuming the first opening brace as well.

`pBibEntryBody :: Parser [Field]`

The body of a bib entry is a list of fields, separated by a comma and spaces. Finally consume the closing brace when done.

`pBibKey :: Parser String`

a bib key (its reference) is a word containing [A-Za-z0-9].

`parsePreamble :: Parser String`

A parser for the preamble keywords. This is just a string preceded by `preamble and surrounded by braces and quotation marks. Unfortunately no support for` string variables yet.

`pKeyValue :: Parser Field`

A key-value parser. A key is a word, followed by =, and the value is surrounded by quotation marks; if, however, the value is numeric, the quotation marks aren't required.

Chapter 7

ParseBib.ParserUtils

```
module ParseBib.ParserUtils (
    parseFeedback,  pMunch1,  pBraced,  spaces
  ) where
```

This module contains some useful parsing utilities.

`parseFeedback`

`::`	`Parser BibTex`	the parser to use
`->`	`String`	the input
`->`	`Feedback BibTex`	return a BibTex, but also allow Feedback

Parse a given input using some UU-Parsinglib parser, but return feedback in the CCO Feedback monad. Useful for reporting failures and error corrections done by the parser.

`pMunch1`

```
::   (Provides st (a -> Bool, [Char], Char) a, Provides st (Munch a) [a])
=>   (a -> Bool)
```

munch all text match-ing this pred-i-cate

```
->   P st [a]
```

the pMunch1 parser is much like pMunch, except that it only succeeds when it can munch 1 or more characters. It fails on the empty string. Inspired by the library function pMunch, from UU-Parsinglib.

`pBraced :: Parser a -> Parser a`

Parser for parsing anything contained in braces.

`spaces :: Parser String`

A parser which greedily consumes whitespace. Useful between entries or fields.

Chapter 8

ParseBib.Tool

```
module ParseBib.Tool (
    mainFunc,  pipeline,  removeComments,  killComments,  bibTexparser,
    bibTex2Aterm
  ) where
```

`mainFunc :: IO ()`

wrap the pipeline in IO and run.

`pipeline :: Component String String`

the pipeline which carries out the aforementioned steps.

`removeComments :: Component String String`

removeComments splits the input into lines, then processes each line for comments using the killComments function

`killComments :: String -> String`

killComments throws away everything after a `%` on a line of input.

`bibTexparser :: Component String BibTex`

run the parser, parseBib, on the input, using the parseFeedback wrapper.

`bibTex2Aterm :: Component BibTex ATerm`

convert the parsed BibTex tree into an ATerm.