

ABR Haskell Libraries – Interface

Andrew Rock
School of Information and Communication Technology
Griffith University
Nathan, Queensland, 4111, Australia
a.rock@griffith.edu.au

Abstract

This document lists and describes the libraries developed for and common to the various systems I have developed in Haskell¹, hiding the implementation details of all module definitions unless exported.

Contents

1 Introduction	3	9 Data.List	6
2 Installation	3	9.1 Sorting	6
3 Args: Command Line Arguments	3	9.2 Combinatorics	6
3.1 Data types	3	9.3 Bag-like operations	7
3.2 Option detection	3	9.4 Set-like operations	7
3.3 Looking up options	3	9.5 Subsequence operations	7
4 CGI	4	10 Data.NameTable	7
4.1 Mime header	4	10.1 Data types	7
4.2 Document type	4	10.2 Creating a name table	7
4.3 Special character encoding	4	10.3 Looking up by name	7
4.4 HTML elements (generic)	4	10.4 Creating a name array	8
4.5 HTML elements (specific shortcuts)	4	11 Data.Queue	8
4.6 Standards	4	11.1 Data type	8
4.7 CGI inputs	4	11.2 Operations	8
5 Control.Check	4	12 Data.Set	8
5.1 Data type	5	12.1 Data type	8
5.2 Sequencing checks	5	12.2 Operations	8
5.3 Parallel checks	5	12.3 Instances	8
6 Control.List	5	12.3.1 Ord	8
6.1 Backwards map	5	12.3.2 Showing	8
7 Data.BSTree: Balanced Binary Search Tree	5	12.3.3 DeepSeq	8
7.1 BSTree type	5	13 Data.SparseSet	9
7.2 BSTree operations	5	13.1 Data type	9
8 Data.HashTables	6	13.2 Operations	9
8.1 Data types	6	14 Daytime	9
8.2 Creating a new hash table	6	14.1 Data types	9
8.3 Updating an existing hash table	6	14.2 Lexing	9
8.4 Looking up in a hash table	6	14.3 Parsing	9
8.5 Dumping a hash table	6	14.4 Instance declarations	10
		14.4.1 Showing	10
		14.4.2 Arithmetic	10
		14.5 Weekday methods	10
		14.6 Daytime methods	10
		15 Debug.Array	10
		15.1 Functions	10
		16 Debug.IArray	10
		16.1 Functions	10
		17 DeepSeq	11
		17.1 Class Definition	11
		17.2 Infix operator	11
		17.3 Instance Declarations	11
		17.3.1 Simple instances	11
		17.3.2 Tuple instances	11
		17.3.3 List instance	11
		17.3.4 Maybe instance	11
		17.3.5 Either instance	11
		18 DeepSeq.BSTree	11
		18.1 Instance declaration	11

¹ABRHLibs – a personal library of Haskell modules Copyright (C) 2007, 2008, Andrew Rock

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

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

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

19 EPS	11		
19.1 Data types	11	27.14Parser result modifiers	20
19.2 Finalizing to EPS	11	27.15Error reporting	20
19.3 Drawing in BPS	11	27.16Instance declarations	20
19.4 Merging BPS components	11	28 Parser.Lexers	20
19.5 Drawing text	11	28.1 Frequently used lexers	20
19.5.1 Switching fonts efficiently	11	29 Parser.Checks	21
19.5.2 Font metrics	12	29.1 Easy lexer and parser sequencing	21
19.5.3 Font tags	12	30 Playing Cards	21
19.5.4 Font strings	12	30.1 Data types	21
19.5.5 Font blocks	12	30.2 Creating decks	21
19.5.6 Text encoding	12	30.3 Instance declarations	21
19.6 Conveniences	12	31 Poker	21
19.7 Instance declarations	12	31.1 Presorting and grouping	22
19.7.1 EPSDrawable	12	31.2 Categorisation of hands	22
20 Geometry	13	31.3 Comparison of hands	22
20.1 Data types	13	31.4 Hands with more than 5 cards	22
20.2 Geometric computations	13	32 QuineMcClusky	22
20.3 Instance declarations	13	32.1 Data types	22
21 Graphs	13	32.2 Simplification	22
21.1 Graph abstract data type	13	32.3 Instance declarations	22
21.2 Sparse graph type	14	32.3.1 DeepSeq	22
21.3 Graph Operations	14	33 Relational Database	22
21.4 Reachability	14	33.1 Definitions and data types	22
21.5 Cycles detection	14	33.2 Relational algebra	23
22 HaskellLexer	14	33.2.1 Projection of a tuple on a relation schema	23
22.1 Handling literate scripts	14	33.2.2 Projection of a relation on a relation schema	23
22.2 Lexing scripts	14	33.2.3 Natural join	23
22.3 Handling the offside rule	14	33.2.4 Union	23
22.4 Diagnostics	14	33.2.5 Difference	23
22.5 Poor man's parsing	14	33.2.6 Selection	23
23 LockFile	15	33.2.7 Renaming	23
23.1 Basic lock operations	15	33.3 Datum operations	23
23.2 Multiple file operations	15	34 SendMail	23
23.3 Guards	15	34.1 Function	24
24 MySQL C API Binding	15	35 Showing	24
24.1 API Data types	15	35.1 Adding Delimiters	24
24.1.1 Basics	15	35.2 Controlling Precision	24
24.1.2 Connections	15	36 Supplies	24
24.1.3 Results	15	36.1 Data types	24
24.1.4 Rows	15	36.2 Creating a supply	24
24.1.5 Fields	15	36.3 Extracting values from a supply	24
24.1.6 Options	15	37 Text.Configs	24
24.2 API Functions	15	37.1 Data types	24
25 MySQL Haskell API	17	37.2 Lexer	24
25.1 Data types	17	37.3 Parser	25
25.1.1 Connections	17	37.4 Showing	25
25.2 Functions	17	37.5 Reading	25
25.2.1 Establishing a connection	17	37.6 Accessing	25
25.2.2 Closing a connection	18	37.6.1 configPaths	25
25.2.3 Issuing a query	18	37.6.2 Lookup functions	26
25.2.4 Fetching query results	18	37.7 Templates	26
26 Parser.Pos	18	37.7.1 Simple Template Markup Language	26
26.1 Positions in a source	18	37.7.2 Populating templates	26
26.2 Overloaded projector	18	38 Text.Markup	26
26.2.1 Container instances	18	38.1 Making text safe for HTML	26
26.3 Relative positions	18	38.2 Making text safe for LaTeX	26
27 Parsing	18	38.3 Converting LaTeX to HTML	26
27.1 Error messages	18	39 Text.String	26
27.2 Results	18	39.1 Word wrapping	26
27.3 Analysers	18	39.2 Justification	27
27.4 Elementary analysers	18	39.3 Tables with justified columns	27
27.5 Elementary analyser combinators	19	39.4 Fields	27
27.6 Analyser result modifiers	19	39.5 Whitespace	27
27.7 More analyser combinators	19	39.6 Pattern matching and substitution	27
27.8 Lexers	19	39.7 Numbers	27
27.9 Elementary lexers	19	39.8 Names	27
27.10Special combinators for lexers	19	39.9 Path catenation operators	27
27.11Frequently used lexers	20		
27.12Parsing	20		
27.13Elementary parsers	20		

39.10 Simple String Delimitation	27
40 Versions	28
40.1 Read the latest version	28
40.2 Date of the latest version	28
40.3 Write the next version	28
40.4 Purge old versions	28
40.5 Remove all versions	28
40.6 Get all versions	28
40.7 Read and write file bottlenecks	28
40.8 Creating and removing directories	28
References	28
Index	28

1 Introduction

This document lists and describes the libraries developed for and common to the various systems I have developed in Haskell. These include ENTRE (a toy functional language interpreter), Virgil (a system for managing groups of students within a course), *Deimos* (an implementation of Defeasible Logic), and *Phobos* (an implementation of Plausible Logic).

Each section in this document, except this introduction, documents the interface to one module from the library.

Haskell code is presented in `typewriter` font. Some text in typewriter font is not Haskell and is boxed to differentiate it from Haskell. The source code for the Haskell modules have been written in the literate style, and the following sections have been produced directly from the Haskell+ \LaTeX source code. The symbol `$` appears in command examples to represent the command line prompt. Multi-line commands are continued with the UNIX escape character, `\`.

The SimpleLit tool has been used to separate interface and implementations into separate \LaTeX documents.

Please let me know of any defects or possible improvements that you spot. Some modules are works-in-progress.

2 Installation

This library is packaged for distribution in the file `ABRHLibs.tar.gz`, available from

<http://www.cit.gu.edu.au/~arock/haskell/>

This file contains a `Makefile`, the library documentation (as two PDF files), and the source code (as a collection of literate sources with the extension `.lit` and the `.lhs` Haskell sources derived from them).

To extract the `.tex` and `.lhs` files from the `.lit` sources, the `SimpleLit` tool is required, and is also available from the link above.

Some modules include parsers, documented with EBNF specifications and syntax diagrams derived from them. The `Syntrax` tool that does this conversion is also available from the link above.

Before compiling, change your current working directory to `ABRHLibs/src`.

```
$ cd ABRHLibs/src
```

To compile the Haskell libraries, `ghc` is required. To typeset the documentation, the tools `bibtex` and `pdflatex` are required. Both are included in the `teTeX` package.

To compile all of the libraries (except those requiring MySQL), use

```
$ make objects
```

To typeset the documentation, use

```
$ make doc
```

To build everything listed above, use

```
$ make all
```

or just

```
$ make
```

To delete intermediate files, use

```
$ make clean
```

To delete those and the objects, interfaces and intermediate \LaTeX and Haskell sources, use

```
$ make CLEAN
```

To rebuild the file `ABRHLibs.tar.gz`, use

```
$ make distribute
```

3 Args: Command Line Arguments

Module `ABR.Args` provides a way to pick apart the meanings of command line arguments.

```
module ABR.Args (
    OptSpec(..), OptVal(..), Options, findOpts,
    lookupFlag, lookupParam, lookupQueue
) where
```

3.1 Data types

An `OptSpec` is used to specify the types of option expected.

```
data OptSpec =
    FlagS String | ParamS String | QueueS String
    deriving (Eq, Show)
```

An option is one of:

- a flag to be set or unset. Specify with `FlagS name`. Users set or unset with `+name` or `-name` respectively.
- a parameter with a value. Specify with `ParamS name`. Users provide values with `-name value`.
- a parameter that can have multiple values. The order of the multiple values might be significant. In this case a queue of strings should be returned. Specify with `QueueS name`. Users provide values with `-name value1 -name value2 ...`

An `OptVal` is used to indicate presence of a command line option. Flags might be `FlagPlus` or `FlagMinus`. Parameters will either return the `ParamValue value` or an indication that the value was missing, `ParamMissingValue`. Queue parameters return `ParamQueue queue`. Missing values for queue parameters might yield an empty queue.

```
data OptVal =
    FlagPlus | FlagMinus |
    ParamValue String | ParamMissingValue |
    ParamQueue (Queue String)
    deriving Show
```

An `Options` is used to map from an option name to its value(s).

```
type Options = BSTree String OptVal
```

3.2 Option detection

`findOpts optSpecs args` returns `(options, leftovers)`, where:

`optSpecs` is a list of option specifications; `args` is a list of command line arguments; `options` is a `BSTree` mapping the option names to the values found; and `leftovers` is a list of any unconsumed arguments, typically file names.

```
findOpts :: [OptSpec] -> [String] -> (Options, [String])
```

3.3 Looking up options

`lookupFlag name options def` returns the value (`FlagPlus` means `True`) stored for the `name`ed flag in `options` or `def` if it has not been properly specified.

```
lookupFlag :: String -> Options -> Bool -> Bool
```

`lookupParam name options def` returns the value stored for the `name`ed parameter in `options` or `def` if it has not been properly specified.

```
lookupParam :: String -> Options -> String -> String
```

`lookupQueue` *name options* returns the list stored for the *named* queue parameter in *options* or [] if it has not been properly specified.

```
lookupQueue :: String -> Options -> [String]
```

4 CGI

Module `ABR.CGI` implements support for Common Gateway Interface (CGI) programming.

```
module ABR.CGI (
  mimeHeader, printMimeHeader, docType, printDocType,
  put, put', HTag, HAttributes, baseE_, isindexE_,
  linkE_, metaE_, nextidE_, inputE_, hrE_, brE_, imgE_,
  isindexN_, hrN_, brN_, htmlE, headE, titleE, styleE,
  bodyE, addressE, blockquoteE, formE, selectE,
  optionE, dlE, dtE, ddE, olE, ulE, dirE,
  menuE, liE, pE, preE, aE, mapE, areaE, citeE,
  codeE, emE, kbdE, sampE, strongE, varE, bE,
  iE, ttE, uE, tableE, captionE, trE, thE, tdE,
  divE, subE, supE, centerE, fontE, smallE,
  bigE, textareaE, h1E, h2E, h3E, h4E, h5E,
  h6E, htmlN, headN, titleN, bodyN, addressN,
  blockquoteN, dlN, dtN, ddN, olN, ulN, dirN,
  menuN, liN, pN, preN, citeN, codeN, emN,
  kbdN, sampN, strongN, varN, bN, iN, ttN, uN,
  tableN, captionN, trN, thN, tdN, subN, supN,
  centerN, smallN, bigN, h1N, h2N, h3N, h4N,
  h5N, h6N, htmlT, htmlError, getQUERY_STRING,
  getPATH_INFO, getSCRIPT_NAME,
  getScriptDirectory, getCONTENT_LENGTH,
  getFormData, getFormData', dumpFormData
) where
```

4.1 Mime header

First things first. A CGI tool should print the magic lines identifying the output as HTML. `mimeHeader` is the MIME header text, which is printed by `printMimeHeader`.

```
mimeHeader :: String
printMimeHeader :: IO ()
```

4.2 Document type

Identify the kind of html being generated. `docType` is printed by which is printed by `printDocType`.

```
docType :: String
printDocType :: IO ()
```

4.3 Special character encoding

`put cs` prints *cs* with all special characters encoded. `put' cs` encodes all control characters.

```
put, put' :: String -> IO ()
```

4.4 HTML elements (generic)

HTML elements have a name (a `HTag`).

```
type HTag = String
```

HTML elements can have a list of attributes (`HAttributes`) of the form *name=value*.

```
type HAttributes = [(String,String)]
```

4.5 HTML elements (specific shortcuts)

This is not an exhaustive list. Add more as needed.

`tagE_ attributes` prints an empty element its *tag* and *attributes*.

```
baseE_, isindexE_, linkE_, metaE_, nextidE_,
inputE_, hrE_, brE_, imgE_
:: HAttributes -> IO ()
```

`tagN_` prints an empty element with its *tag* and no attributes.

```
isindexN_, hrN_, brN_
:: IO ()
```

`tagE attributes contents` prints a non-empty element with its *tag*, *attributes* and *contents*.

```
htmlE, headE, titleE, styleE, bodyE, addressE, blockquoteE,
formE, selectE, optionE, dlE, dtE, ddE, olE,
ulE, dirE, menuE, liE, pE, preE, aE, mapE,
areaE, citeE, codeE, emE, kbdE, sampE, strongE,
varE, bE, iE, ttE, uE, tableE, captionE, trE,
thE, tdE, divE, subE, supE, centerE, fontE,
smallE, bigE, textareaE, h1E, h2E, h3E, h4E,
h5E, h6E
:: HAttributes -> IO () -> IO ()
```

`tagN contents` prints a non-empty element with its *tag*, *contents* and no attributes.

```
htmlN, headN, titleN, bodyN, addressN, blockquoteN,
dlN, dtN, ddN, olN, ulN, dirN, menuN, liN, pN,
preN, citeN, codeN, emN, kbdN, sampN, strongN,
varN, bN, iN, ttN, uN, tableN, captionN, trN,
thN, tdN, subN, supN, centerN, smallN, bigN,
h1N, h2N, h3N, h4N, h5N, h6N
:: IO () -> IO ()
```

4.6 Standards

The `html` element needs to have certain attributes to meet standards. `htmlT` applies the attributes to go with the transitional doctype above.

```
htmlT :: IO () -> IO ()
```

4.7 CGI inputs

`getQUERY_STRING` returns the text after the ? in a URL.

`getPATH_INFO` returns the extra path info after the name of the CGI tool. `getSCRIPT_NAME` returns the URL of the CGI tool. `getScriptDirectory` returns the URL of the directory the CGI binary is in.

```
getQUERY_STRING, getPATH_INFO, getSCRIPT_NAME,
getScriptDirectory :: IO String
```

`getCONTENT_LENGTH` returns the number of bytes of content arriving via standard input.

```
getCONTENT_LENGTH :: IO Int
```

`getFormData` reads standard input to obtain the post method inputs, decodes them and returns them in a binary search tree.

```
getFormData :: IO (BSTree String String)
```

`getFormData'` reads standard input to obtain the post method inputs in the `enctype="multipart/form-data"` format, decodes them and returns them in a binary search tree.²

```
getFormData' :: IO (BSTree String String)
```

For debugging forms: `dumpFormData tree` outputs the form inputs in a nicely encoded XHTML fragment.

```
dumpFormData :: BSTree String String -> IO ()
```

²This function in part by Annie Lo, 2134CIT Programming Paradigms and Languages, Advanced Studies project, 2004.

5 Control.Check

Module `ABR.Control.Check` implements checks as operations to be performed that may succeed or fail. Checks are often performed in a sequence. Composing lots of checks can lead to big, ugly cascades of case expressions. This module provides a way to do it more compactly and neatly.³

```
module ABR.Control.Check (
  CheckResult(..), Check, (&?), (+?), (??), (*?)
) where
infixl 2 &?, +?, #?, ??, *?
```

5.1 Data type

The result of a `Check`, a `CheckResult` is either a `CheckPass` with the correct result, or a `CheckFail` with some alternate data, probably an error message string.

```
data CheckResult passType failType = CheckPass passType
                                     | CheckFail failType
  deriving (Eq, Ord, Show)
```

A `Check` takes some object and returns a `CheckResult`.

```
type Check objectType passType failType
  = objectType -> CheckResult passType failType
```

5.2 Sequencing checks

`c1 &? c2` sequence composes check `c1` and `c2` in that order. `c1` is applied first. If it succeeds, then `c2` is applied to the result.

```
(&?) :: Check a b d -> Check b c d -> Check a c d
```

5.3 Parallel checks

`r1 +? r2` combines check results `r1` and `r2`. If both results are passes only `r2` is returned. If only one result is a pass, then the other failing result is returned. If both are fails, then a fail is returned with the catenation of the error messages. This leads to the restriction that the fail data type must be a list type.

```
(+?) :: CheckResult a [b] -> CheckResult a [b]
      -> CheckResult a [b]
```

`c #? xs` applies check `c` to each of the elements of `xs` in parallel, returning only the last result if all checks pass or all of the error messages catenated if any checks fail.

```
(#?) :: Check a b [c] -> Check [a] b [c]
```

`cs ?? x` applies all the checks in `cs` to `x` in parallel; returning only the last result if all checks pass or all of the error messages catenated if any checks fail.

```
(??) :: [Check a b [c]] -> Check a b [c]
```

`cs *? xs` applies all the checks in `cs` to all of the elements of `xs` in parallel; returning only the last result (the last check applied to the last thing) if all checks pass or all of the error messages catenated if any checks fail.

```
(*?) :: [Check a b [c]] -> Check [a] b [c]
```

6 Control.List

Module `ABR.Control.List` implements control abstractions involving lists.

```
module ABR.Control.List (pam) where
```

6.1 Backwards map

`pam fs x` returns the list of results obtained by applying all the functions in `fs` to `x`.

```
pam :: [a -> b] -> a -> [b]
```

³Thanks to Daniel Young for suggested extensions to this module.

7 Data.BSTree: Balanced Binary Search Tree

Module `ABR.Data.BSTree` implements a depth/height balanced (AVL) binary search tree abstract data type.

```
module ABR.Data.BSTree (
  BSTree(..), emptyBST, nullBST, depthBST, updateBST,
  deleteBST, lookupBST, memberBST, lookupGuard,
  flattenBST, domBST, ranBST, countBST, leftBST,
  rightBST, mapBST, pairs2BST, list2BST
) where
```

7.1 BSTree type

A `BSTree` is either empty or a node containing a key, an associated value and left and right sub-trees. Type `key` must be an instance of type class `Ord`, so that `<` and `==` work.

```
data (Ord key) => BSTree key value =
```

All the functions in this module maintain the following invariant: The depth of left and right sub-trees differ by no more than 1.

7.2 BSTree operations

`emptyBST` is an empty `BSTree`.

```
emptyBST :: Ord k => BSTree k v
```

`nullBST t` returns `True` iff `t` is empty.

```
nullBST :: Ord k => BSTree k v -> Bool
```

`depthBST t` returns the depth of `a`.

```
depthBST :: Ord k => BSTree k v -> Int
```

`updateBST f key value bst` returns the new tree obtained by updating `bst` with the `key` and `value`. If the `key` already exists, `f` is used to combine the two values. Use `(\x _ -> x)` to merely replace.

```
updateBST :: Ord k => (v -> v -> v) -> k -> v
          -> BSTree k v -> BSTree k v
```

`deleteBST k t` returns the new tree obtained by deleting the `k` and its associated value from `t`.

```
deleteBST :: Ord k => k -> BSTree k v -> BSTree k v
```

`lookupBST k t` returns `Just v`, where `v` is the value associated with `k` in `t`, or `Nothing`.

```
lookupBST :: Ord k => k -> BSTree k v -> Maybe v
```

`memberBST k t` returns `True` iff `k` occurs in `t`.

```
memberBST :: Ord k => k -> BSTree k v -> Bool
```

`lookupGuard bst keys handler process` tries to look up the `keys`. If any are missing the `handler` is applied to the first missing key otherwise the `process` is applied to the list of values successfully looked up.

```
lookupGuard :: Ord a => BSTree a b -> [a] -> (a -> c)
          -> ([b] -> c) -> c
```

`flattenBST t` returns the list of tuples (k, v) in `t` in ascending order of key.

```
flattenBST :: Ord k => BSTree k v -> [(k,v)]
```

`domBST t` returns the list of keys in `t` in ascending order of key.

```
domBST :: Ord k => BSTree k v -> [k]
```

`domBST t` returns the list of values in `t` in ascending order of key.

```
ranBST :: Ord k => BSTree k v -> [v]
```

`pairs2BST kvs` converts an association list `kvs` of pairs (k, v) to a `BSTree`. If there are duplicate `v`'s for a `k`, only the first is retained.

```
pairs2BST :: Ord k => [(k,v)] -> BSTree k v
```

`list2BST ks v` converts a list of keys `ks` to a `BSTree`. The values in the tree are all assigned `v`.

```
list2BST :: Ord k => [k] -> v -> BSTree k v
```

countBST *t* returns the number of elements in *t*.

```
countBST :: Ord k => BSTree k v -> Int
```

leftBST *t* returns the left-most element of *t*. **rightBST** *t* returns the right-most element of *t*.

```
leftBST, rightBST :: Ord k => BSTree k v -> Maybe (k, v)
rightBST Empty = Nothing
rightBST (Node k v _ Empty _) = Just (k, v)
rightBST (Node k v _ r _) = rightBST r
```

mapBST *f t* returns the tree formed by applying *f* to all of the values in *t*. The keys are not changed.

```
mapBST :: Ord k => (v -> v') -> BSTree k v -> BSTree k v'
```

8 Data.HashTables

Module **ABR.Data.HashTables** implements hash tables in as efficient a manner as I can, while retaining as much polymorphism as possible. The efficiency is made possible by exploiting the mutable arrays built into the IO monad.

```
module ABR.Data.HashTables (
  HashTable, newHT, updateHT, lookupHT, dumpHT
) where
```

8.1 Data types

A **HashTable** is a mapping from keys to associated values. Access is speeded by distributing the values across an array that can be accessed in constant time using a hashing function to map the keys to index values.

```
type HashTable key index value =
```

8.2 Creating a new hash table

newHT (*lo, hi*) returns a new empty hash table, where (*lo, hi*) is the bounds on the array and therefore the range of the hashing function.

```
newHT :: (Ix ix, Ord key) =>
  (ix,ix) -> IO (HashTable key ix value)
```

8.3 Updating an existing hash table

updateHT *hashFun updateFun ht k v* updates the hash table *ht* with the key *k* and associated value *v*. The function *hashFun* maps keys to hashing values. The function *updateFun* is used to combine the new value *v* with any existing value already associated with this *key*. Use $(\backslash x _ \rightarrow x)$ to merely replace the old value.

```
updateHT :: (Ix ix, Ord key) =>
  (key -> ix) -> (value -> value -> value) ->
  HashTable key ix value -> key -> value -> IO ()
```

8.4 Looking up in a hash table

lookupHT *hashFun k ht* returns **Just** *v*, where *v* is the value associated with *k* in the hash table *ht*. If *k* is not in the hash table, **Nothing** is returned. The function *hashFun* maps keys to hashing values.

```
lookupHT :: (Ix ix, Ord key) =>
  (key -> ix) -> key -> HashTable key ix value ->
  IO (Maybe value)
```

8.5 Dumping a hash table

dumpHT *ht* prints the hash table *ht* in a fairly crude format, adequate for assessment of the hashing function.

```
dumpHT ::
  (Ix ix, Enum ix, Ord key, Show key, Show value) =>
  HashTable key ix value -> IO ()
```

9 Data.List

Module **ABR.Data.List** is a collection of functions that operate on lists.

```
module ABR.Data.List (
  merge, msort, split, cartProd, interleave, separate,
  fragments, fragments', dropEach, permutations,
  permutations', combinations, subBag, bagElem,
  powSet, powSet_ge1, powSet', powSet_ge1',
  properSublists, pPlus, meet, disjoint, allUnique,
  duplicates, snub, (+:), msub, isSubset, findSubset,
  noSuperSets, isSubSequence, notSubSequence, chop,
  chops, subsSuffix, odiff, osect, onion,
  sortByLength, trimN, trim2
) where
```

9.1 Sorting

msort *lt xs* sorts *xs* using *lt* as the less-than operator.

```
msort :: (a -> a -> Bool) -> [a] -> [a]
```

sortByLength *xss* sorts a list of lists into non-descending order of length.

```
sortByLength :: [[a]] -> [[a]]
```

9.2 Combinatorics

merge *lt xs ys* merges lists *xs* and *ys* preserving the non-descending order in *xs* and *ys* using *lt* to decide what is less than what.

```
merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
```

split *xs* splits *xs* into two lists of the alternate elements of *xs*.

```
split :: [a] -> ([a],[a])
```

cartProd produces the cartesian product of an arbitrary number of lists. That is, **cartProd** [*xs*₁, *xs*₂, ...] returns *xs*₁ × *xs*₂ × ... Note: **Prelude.sequence** can be used to do the same job.

```
cartProd :: [[a]] -> [[a]]
```

interleave *x xs* returns the list of lists formed by inserting *x* in each possible place in *xs*.

```
interleave :: a -> [a] -> [[a]]
```

separate *xs yss* concatenates *yss*, with *xs* interspersed.

```
separate :: [a] -> [[a]] -> [a]
```

fragments *xs* returns the list of fragments

(*beforeElems*, *elem*, *afterElems*) for each element of *xs*. The elements in *beforeElems* are in reverse order with respect to *xs*.

```
fragments :: [a] -> [[([a],a,[a])]
```

fragments' *xs* returns the list of fragments (*elem*, *otherElems*) for each element of *xs*. The elements in *otherElems* are in no particular order.

```
fragments' :: [a] -> [(a,[a])]
```

dropEach *xs* returns the list of lists obtained by deleting each element of *xs*.

```
dropEach :: [a] -> [[a]]
```

permutations *k xs* returns all the permutations of *k* elements selected from *xs*. Precondition: $0 \leq k \leq \text{length } xs$.

```
permutations :: Int -> [a] -> [[a]]
```

permutations' *xs* returns all the permutations of the elements of *xs*.

```
permutations' :: [a] -> [[a]]
```

combinations *k xs* returns all the combinations of *k* elements drawn from *xs*. Precondition: $0 \leq k \leq \text{length } xs$.

```
combinations :: Int -> [a] -> [[a]]
```

powSet *xs* returns the list of sub-lists of *xs*. This version does not return them in an order that monotonically increases in length.

`powSet :: [a] -> [[a]]`

`powSet_ge1` *xs* returns the list of sub-lists of *xs* with at least 1 element. This version does not return them in an order that monotonically increases in length.

`powSet_ge1 :: [a] -> [[a]]`

`powSet'` *xs* returns the list of sub-lists of *xs*. This version *does* return them in an order that monotonically increases in length.

`powSet' :: [a] -> [[a]]`

`powSet_ge1'` *xs* returns the list of sub-lists of *xs* with at least 1 element. This version *does* return them in an order that monotonically increases in length.

`powSet_ge1' :: [a] -> [[a]]`

`properSublists` *xs* returns the list of strict sub-lists of *xs* with at least 1 element, in an order that monotonically increases in length.

`properSublists :: [a] -> [[a]]`

9.3 Bag-like operations

xs 'subBag' *ys* returns True iff every element of *xs* occurs at least as frequently in *ys* as it does in *xs*.

`subBag :: Eq a => [a] -> [a] -> Bool`

`bagElem` *xs xss* returns True iff *xs* or some permutation of *xs* is an element of *xss*.

`bagElem :: Eq a => [a] -> [[a]] -> Bool`

9.4 Set-like operations

`allUnique` *xs* returns True iff all elements of *xs* are unique.

`allUnique :: Eq a => [a] -> Bool`

`duplicates` *xs* returns all of the elements of *xs* that are duplicated.

`duplicates :: Eq a => [a] -> [a]`

`snuB` *xs* returns the unique elements of *xs* in non-descending order, and does it in $O(N \log N)$ time.

`snuB :: Ord a => [a] -> [a]`

`X +:` *x* returns $X \cup \{x\}$.

`infixl 5 +:`

`(+:) :: Ord a => [a] -> a -> [a]`

`mnuB` *xs ys* merges lists *xs* and *ys* which must be in strictly ascending order. Any elements that occur in *xs* and *ys* occur only once in the result.

`mnuB :: Ord a => [a] -> [a] -> [a]`

`isSubset` *xs ys* returns True iff $xs \subseteq ys$. Precondition: *xs* and *ys* are in strictly ascending order.

`isSubset :: (Ord a) => [a] -> [a] -> Bool`

`isProperSubset` *xs ys* returns True iff $xs \subset ys$. Precondition: *xs* and *ys* are in strictly ascending order.

`isProperSubset :: (Ord a) => [a] -> [a] -> Bool`

`odiff` *xs ys* returns the set difference $xs - ys$. Precondition: *xs* and *ys* are in strictly ascending order.

`odiff :: (Eq a, Ord a) => [a] -> [a] -> [a]`

`osect` *xs ys* returns the set intersection $xs \cap ys$. Precondition: *xs* and *ys* are in strictly ascending order.

`osect :: (Eq a, Ord a) => [a] -> [a] -> [a]`

`ounion` *xs ys* returns the set union $xs \cup ys$. Precondition: *xs* and *ys* are in strictly ascending order.

`ounion :: (Eq a, Ord a) => [a] -> [a] -> [a]`

`findSubset` *xs ys* returns Just *xs* if $xs \subseteq ys$ or Just *ys* if $ys \subseteq xs$, otherwise Nothing. Precondition: *xs* and *ys* are in strictly ascending order.

`findSubset :: Ord a => [a] -> [a] -> Maybe [a]`

`noSuperSets` *xss* reduces *xss* by removing all elements of *xss* that are supersets of any other elements of *xss*. Preconditions: All elements of *xss* are in strictly ascending order; and all elements of *xss* are unique.

`noSuperSets :: Ord a => [[a]] -> [[a]]`

`disjoint` *A B* returns True iff $A \cap B = \{\}$.

`disjoint :: Eq a => [a] -> [a] -> Bool`

`meet` *A B* returns True iff $A \cap B \neq \{\}$.

`meet :: Eq a => [a] -> [a] -> Bool`

`pPlus` *L H* returns $\mathcal{P}^+(L, H) = \{K : K \subseteq L \text{ and } K \neq \{\} \text{ and } H \cap K = \{\}\}$.

`pPlus :: Eq a => [a] -> [a] -> [[a]]`

9.5 Subsequence operations

`isSubSequence` *ps cs* returns True iff *ps* is a sequence that occurs in *cs*. This implementation uses only a brute force algorithm, $O(m \times n)$, for $m = \text{length } ps$ and $n = \text{length } cs$.

`isSubSequence :: Eq a => [a] -> [a] -> Bool`

`notSubSequence` *ps cs* returns True iff *ps* is not a sequence that occurs in *cs*.

`notSubSequence :: Eq a => [a] -> [a] -> Bool`

`chop` *x xs* returns the sublists in *xs* that are separated by elements equal to *x*.

`chop :: Eq a => a -> [a] -> [[a]]`

`chops` *bs xs* returns the sublists in *xs* that are separated by sequences equal to *bs*.

`chops :: Eq a => [a] -> [a] -> [[a]]`

`subsSuffix` *sep suf xs* replaces anything after the rightmost occurrence of *sep* in *xs* with *suf*. If *suf* does not occur in *xs*, then *sep* and *suf* are appended. Use this to create output file names from input file names.

`subsSuffix :: Eq a => a -> [a] -> [a] -> [a]`

`trimN` *n xs* drops *n* elements from both ends of a list.

`trimN :: Int -> [a] -> [a]`

`trim2` *xs* drops 2 elements from both ends of a list. Useful for comments and the like.

`trim2 :: [a] -> [a]`

10 Data.NameTable

Module `ABR.Data.NameTable` implements structures for the efficient accumulation of names, assigning unique, sequential integers to each name and mapping between them.

```
module ABR.Data.NameTable (  
    NameTable, newNameTable, insertName, lookupName,  
    NameArray, makeNameArray  
) where
```

10.1 Data types

A `NameTable` is a triple of:

- a hash table mapping strings to integers;
- a supply of sequential integers starting from 0;
- a hashing function.

`type NameTable =`

For the reverse mapping an array of names, a `NameArray`, is optimal.

`type NameArray = Array Int String`

10.2 Creating a name table

`newNameTable` *size* creates a `NameTable` of a given *size* (which should be prime).

```
newNameTable :: Int -> IO NameTable
```

`insertName` *t n* inserts name *n* into the name table *t*. If the name already exists, nothing happens. If the name is new, it is added to the table and assigned the next sequence number.

```
insertName :: NameTable -> String -> IO ()
```

10.3 Looking up by name

`lookupName` *t n* retrieves the sequence number for the given name *n* in name table *t*, provided it exists.

```
lookupName :: NameTable -> String -> IO (Maybe Int)
```

10.4 Creating a name array

`makeNameArray` *t* builds an array for mapping sequence numbers back to names.

```
makeNameArray :: NameTable -> IO NameArray
```

11 Data.Queue

The `ABR.Data.Queue` module implements the Queue ADT.

```
module ABR.Data.Queue (
  Queue, emptyQ, isEmptyQ, attachQ, frontQ, detachQ,
  extractQ
) where
```

11.1 Data type

A `Queue` is a first-in-first-out sequence.

```
type Queue a =
```

11.2 Operations

`emptyQ` is an empty queue.

```
emptyQ :: Queue a
```

`isEmptyQ` *q* returns `True` iff queue *q* is empty.

```
isEmptyQ :: Queue a -> Bool
```

`attachQ` *e q* attaches *e* to the back of queue *q*.

```
attachQ :: a -> Queue a -> Queue a
```

`frontQ` *q* returns the value at the front of queue *q*.

```
frontQ :: Queue a -> a
```

`detachQ` *q* returns the element that was at the front of queue *q* and the *q* after that element has been detached.

```
detachQ :: Queue a -> (a, Queue a)
```

`extractQ` *q* returns the list of all elements in queue *q*.

```
extractQ :: Queue a -> [a]
```

12 Data.Set

Module `ABR.Data.Set` implements a set type where the elements are orderable, but selected from too large a domain to make an array implementation practical. The sets are implemented with a list.

```
module ABR.Data.Set (
  Set, eset, set, set1, unset1, list, (.|), (.&), (.-),
  (.), (.-), (.<), (.<=), card, smap, snull, select,
  (.*), sprod, (.*), sany, sall, sfoldl, sfoldl1,
  sfoldr, sfoldr1, sunion, ssect
) where
```

12.1 Data type

```
data Set a =
```

12.2 Operations

```
infixl 7 .&, .*, .*.
```

```
infixl 6 .|, .-, .+
```

```
infix 5 .<-, .<, .<=
```

`eset` is `{}`.

```
eset :: Set a
```

`set` *xs* returns the set of elements in *xs*.

```
set :: (Ord a) => [a] -> Set a
```

`set1` *x* returns `{x}`.

```
set1 :: a -> Set a
```

`unset1` `{x}` returns *x*.

```
unset1 :: Set a -> a
```

`list` *A* returns the list of elements in *A*.

```
list :: Set a -> [a]
```

A `.|` *B* returns $A \cup B$. *A* `.&` *B* returns $A \cap B$. *A* `.-` *B* returns $A - B$.

```
(.|), (.&), (.-) :: (Ord a) => Set a -> Set a -> Set a
```

A `.+` *x* returns $A \cup \{x\}$.

```
(.+), (.-) :: (Ord a) => Set a -> a -> Set a
```

x `.<-` *A* returns `True` iff $x \in A$.

```
(.<-) :: (Ord a) => a -> Set a -> Bool
```

A `.<` *B* returns `True` iff $A \subset B$. *A* `.<=` *B* returns `True` iff $A \subseteq B$.

```
(.<), (.<=) :: (Ord a) => Set a -> Set a -> Bool
```

`card` *A* returns $|A|$.

```
card :: Set a -> Int
```

`smap` *f A* returns $\{fx : x \in A\}$.

```
smap :: (Ord b) => (a -> b) -> Set a -> Set b
```

`snull` *A* returns `True` iff $A = \{\}$.

```
snull :: Set a -> Bool
```

`select` *P A* returns $\{x : x \in A \text{ and } P(x)\}$.

```
select :: (a -> Bool) -> Set a -> Set a
```

A `.*` *B* returns $A \otimes B = \otimes\{A, B\}$.

```
(.*) :: Ord a => Set a -> Set a -> Set (Set a)
```

`sprod` $\{A_1, \dots, A_n\}$ returns $\otimes\{A_1, \dots, A_n\}$, where $\otimes\{\} = \{\{\}\}$,

$\otimes\{A\} = \{\{a\} : a \in A\}$, and $\otimes\{A_1, \dots, A_n\} = A_1 \otimes \dots \otimes A_n = \{\{a_1, \dots, a_n\} : (a_1, \dots, a_n) \in A_1 \times \dots \times A_n\}$

```
sprod :: Ord a => Set (Set a) -> Set (Set a)
```

A `.*` *B* returns $A \times B$.

```
(.*) :: (Ord a, Ord b) =>
  Set a -> Set b -> Set (a,b)
```

`sall` *P A* returns `True` iff $\forall a \in A, P(a)$. `sany` *P A* returns `True` iff $\exists a \in A, P(a)$.

```
sall, sany :: (a -> Bool) -> Set a -> Bool
```

`sfoldl`, `sfoldr`, `sfoldl1` and `sfoldr1` are analogues of `foldl`, `foldr`, `foldl1` and `foldr1` respectively.

```
sfoldl :: (a -> b -> a) -> a -> Set b -> a
```

```
sfoldr :: (a -> b -> b) -> b -> Set a -> b
```

```
sfoldl1, sfoldr1 :: (a -> a -> a) -> Set a -> a
```

`sunion` $\{A_1, \dots, A_n\}$ returns $\bigcup\{A_1, \dots, A_n\} = A_1 \cup \dots \cup A_n$.

`ssect` $\{A_1, \dots, A_n\}$ returns $\bigcap\{A_1, \dots, A_n\} = A_1 \cap \dots \cap A_n$.

```
sunion, ssect :: Ord a => Set (Set a) -> Set a
```

12.3 Instances

12.3.1 Ord

instance (Ord a) => Ord (Set a) where

12.3.2 Showing

instance (Show a) => Show (Set a) where

12.3.3 DeepSeq

instance (DeepSeq a) => DeepSeq (Set a) where

13 Data.SparseSet

Module `ABR.Data.SparseSet` implements a set type where the elements are orderable, but too selected from too large a domain to make an array implementation practical.

```
module ABR.Data.SparseSet (
  SparseSet, emptySS, nullSS, insertSS, mkSS,
  deleteSS, elemSS, notElemSS, flattenSS, list2SS,
  countSS, isSubSet, unionSS, sectSS, diffSS
) where
```

13.1 Data type

A `SparseSet` is implemented with a height-balanced tree.

```
type SparseSet a =
```

13.2 Operations

`emptySS` is `{}`.

```
emptySS :: Ord k => SparseSet k
```

`nullSS` `s` returns `True` iff `s = {}`.

```
nullSS :: Ord k => SparseSet k -> Bool
```

`insertSS` `e s` returns `{e} ∪ s`.

```
insertSS :: Ord k => k -> SparseSet k -> SparseSet k
```

`mkSS` `e` returns `{e}`.

```
mkSS :: Ord k => k -> SparseSet k
```

`deleteSS` `s e` returns `s - {e}`.

```
deleteSS :: Ord k => SparseSet k -> k -> SparseSet k
```

`elemSS` `s` returns `True` iff `e ∈ s`.

```
elemSS :: Ord k => k -> SparseSet k -> Bool
```

`notElemSS` `s` returns `True` iff `e ∉ s`.

```
notElemSS :: Ord k => k -> SparseSet k -> Bool
```

`isSubSet` `a b` returns `True` iff `a ⊆ b`.

```
isSubSet :: Ord k => SparseSet k -> SparseSet k -> Bool
```

`flattenSS` `s` returns the list of elements of `s` in ascending order.

```
flattenSS :: Ord k => SparseSet k -> [k]
```

`list2SS` `xs` returns the set of elements in `xs`.

```
list2SS :: Ord k => [k] -> SparseSet k
```

`countSS` `s` returns `|s|`.

```
countSS :: Ord k => SparseSet k -> Int
```

`unionSS` `a b` returns `a ∪ b`. This is faster if `|a| < |b|`.

```
unionSS :: Ord k => SparseSet k -> SparseSet k
         -> SparseSet k
```

`sectSS` `a b` returns `a ∩ b`. This is faster if `|a| < |b|`.

```
sectSS :: Ord k => SparseSet k -> SparseSet k
        -> SparseSet k
```

`diffSS` `a b` returns `a - b`.

```
diffSS :: Ord k => SparseSet k -> SparseSet k
        -> SparseSet k
```

14 Daytime

Module `ABR.Daytime` provides time of day and weekday manipulations.

```
module ABR.Daytime (
  Daytime(..), Weekday(..), daytimeL, weekdayP,
  daytimeP, showDT24, inInterval, tomorrow,
  yesterday
) where
```

14.1 Data types

A `Daytime` consists of: hours, `dtHrs`; minutes, `dtMins`; and seconds, `dtSecs`.

```
data Daytime = Daytime {
  dtHrs  :: Int,
  dtMins :: Int,
  dtSecs :: Int
} deriving (Eq, Ord)
```

A `Weekday` is one of: `Sunday`; `Monday`; `Tuesday`; `Wednesday`; `Thursday`; `Friday`; or `Saturday`.

```
data Weekday =
  Sunday | Monday | Tuesday | Wednesday | Thursday |
  Friday | Saturday
  deriving (Eq, Ord, Enum, Show)
```

14.2 Lexing

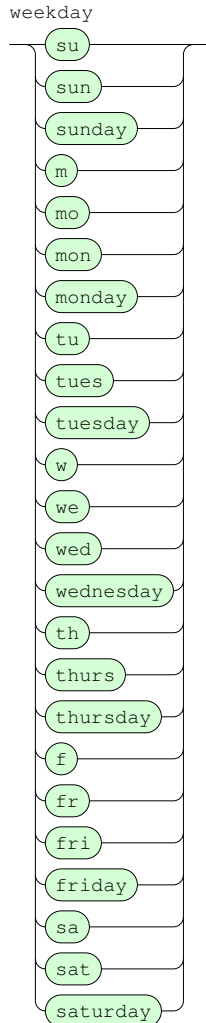
`daytimeL` recognizes the numbers words and symbols that might occur in a day and/or time specification.

```
daytimeL :: Lexer
```

14.3 Parsing

A weekday has this case-insensitive syntax:

```
weekday ::=
  "su" | "sun" | "sunday"
  | "m" | "mo" | "mon" | "monday"
  | "tu" | "tues" | "tuesday"
  | "w" | "we" | "wed" | "wednesday"
  | "th" | "thurs" | "thursday"
  | "f" | "fr" | "fri" | "friday"
  | "sa" | "sat" | "saturday".
```

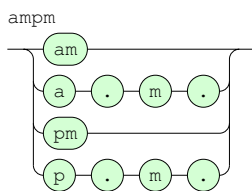


weekdayP parses a Weekday.

`weekdayP :: Parser Weekday`

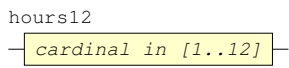
An AM/PM designation has this case-insensitive syntax:

```
ampm ::= "am" | "a" "." "m" "."
       | "pm" | "p" "." "m" ".".
```

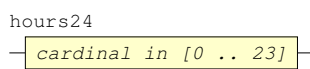


The hours in a daytime are either in 12 or 24 hour formats. Minutes and seconds are preceded by either a colon or a period and are between 0 and 59.

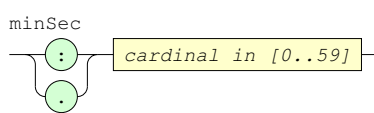
```
hours12 ::= $cardinal in [1..12]$.
```



```
hours24 ::= $cardinal in [0 .. 23]$.
```

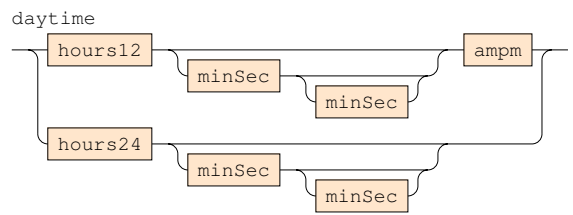


```
minSec ::= (":" | ".") $cardinal in [0..59]$.
```



A daytime has this syntax:

```
daytime ::=
  hours12 [minSec [minSec]] ampm
  | hours24 [minSec [minSec]].
```

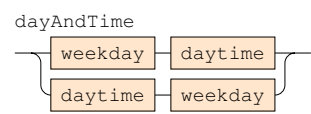


daytimeP parses a daytime.

`daytimeP :: Parser Daytime`

A day and time has this syntax:

```
dayAndTime ::=
  weekday daytime
  | daytime weekday.
```



dayAndTimeP parses a day and a time.

`dayAndTimeP :: Parser (Weekday,Daytime)`

14.4 Instance declarations

14.4.1 Showing

`instance Show Daytime where`

14.4.2 Arithmetic

`instance Num Daytime where`

14.5 Weekday methods

tomorrow *d* returns the weekday after *d*. **yesterday** *d* returns the weekday after *d*.

`tomorrow, yesterday :: Weekday -> Weekday`

14.6 Daytime methods

inInterval *start duration time* returns True iff $start \leq time < start + duration$.

`inInterval :: Daytime -> Daytime -> Daytime -> Bool`

showDT24 *t* shows the daytime *t* in 24 hour format suppressing the seconds.

`showDT24 :: Daytime -> String`

15 Debug.Array

Module **ABR.Debug.Array** is used to help debug programs that use arrays.

`module ABR.Debug.Array (array', accumArray', (!!!)) where`

15.1 Functions

A **!!!** is a replacement for ! that displays a different error message so you can pin down which array indexing operation is out of range.

`(!!!) :: (Ix i, Show i, Show e) => Array i e -> i -> e`

A **array'** is a replacement for array that displays a different error message so you can pin down which array indexing operation is out of range.

`array' :: (Ix i, Show i, Show e) => (i,i) -> [(i,e)] -> Array i e`

A `accumArray'` is a replacement for `accumArray` that displays a different error message so you can pin down which array indexing operation is out of range.

```
accumArray' :: (Ix i, Show i, Show a) => (e -> a -> e)
            -> e -> (i,i) -> [(i,a)] -> Array i e
```

16 Debug.IArray

Module `ABR.Debug.IArray` is used to help debug programs that use immutable arrays.

```
module ABR.Debug.IArray (array', accumArray', (!!!)) where
```

16.1 Functions

A `!!!` is a replacement for `!` that displays a different error message so you can pin down which array indexing operation is out of range.

```
(!!!) :: (IArray a e, Ix i, Show i, Show (a i e)) =>
        a i e -> i -> e
```

A `array'` is a replacement for `array` that displays a different error message so you can pin down which array indexing operation is out of range.

```
array' :: (Ix i, Show i, Show e) =>
         (i,i) -> [(i,e)] -> Array i e
```

A `accumArray'` is a replacement for `accumArray` that displays a different error message so you can pin down which array indexing operation is out of range.

```
accumArray' :: (Ix i, Show i, Show a) => (e -> a -> e)
            -> e -> (i,i) -> [(i,a)] -> Array i e
```

17 DeepSeq

Module `ABR.DeepSeq` was pinched from Dean Herington, who says:

“The prelude support for strict evaluation, `seq` and `($!)`, evaluate only enough to ensure that the value being forced is not bottom. In your case you need a deeper evaluation to be forced.

“A clean (though somewhat tedious) way to achieve what you need is with the `deepSeq` function from the following module.

“The `DeepSeq` class provides a method `deepSeq` that is similar to `seq` except that it forces deep evaluation of its first argument before returning its second argument.

“Instances of `DeepSeq` are provided for Prelude types. Other instances must be supplied by users of this module.”

```
module ABR.DeepSeq (DeepSeq(..), ($!)) where
```

17.1 Class Definition

`DeepSeq` has only one method, `deepSeq` $x y$ deeply evaluates x and then returns y .

```
class DeepSeq a where
    deepSeq :: a -> b -> b
```

17.2 Infix operator

f `$$$` x deeply evaluates x and then returns $f x$.

```
infixr 0 'deepSeq', $$$
($!!) :: (DeepSeq a) => (a -> b) -> a -> b
```

17.3 Instance Declarations

17.3.1 Simple instances

```
instance DeepSeq ()       where {}
instance DeepSeq Bool    where {}
instance DeepSeq Char    where {}
instance DeepSeq Ordering where {}
instance DeepSeq Integer where {}
instance DeepSeq Int     where {}
instance DeepSeq Float   where {}
instance DeepSeq Double  where {}
```

17.3.2 Tuple instances

```
instance (DeepSeq a, DeepSeq b) =>
        DeepSeq (a,b) where
instance (DeepSeq a, DeepSeq b, DeepSeq c) =>
        DeepSeq (a,b,c) where
instance (DeepSeq a,DeepSeq b,DeepSeq c,DeepSeq d) =>
        DeepSeq (a,b,c,d) where
instance (DeepSeq a, DeepSeq b, DeepSeq c, DeepSeq d,
        DeepSeq e) => DeepSeq (a,b,c,d,e) where
instance (DeepSeq a, DeepSeq b, DeepSeq c, DeepSeq d,
        DeepSeq e, DeepSeq f) => DeepSeq (a,b,c,d,e,f) where
instance (DeepSeq a, DeepSeq b, DeepSeq c, DeepSeq d,
        DeepSeq e,DeepSeq f,DeepSeq g) =>
        DeepSeq (a,b,c,d,e,f,g) where
```

17.3.3 List instance

```
instance (DeepSeq a) => DeepSeq [a] where
```

17.3.4 Maybe instance

```
instance (DeepSeq a) => DeepSeq (Maybe a) where
```

17.3.5 Either instance

```
instance (DeepSeq a, DeepSeq b) =>
        DeepSeq (Either a b) where
```

18 DeepSeq.BSTree

Module `ABR.Data.BSTree` implements a depth/height balanced (AVL) binary search tree abstract data type.

```
module ABR.DeepSeq.BSTree where
```

18.1 Instance declaration

```
instance (DeepSeq k, DeepSeq v, Ord k) =>
        DeepSeq (BSTree k v) where
```

19 EPS

Module `ABR.EPS` provides support for the composition of Encapsulated PostScript (EPS).

```
module ABR.EPS (
    EPS, PS, BPS, bpsToEps, EPSDrawable(..),
    joinBPS, setUpFonts, times10Width,
    symbol10Width, FontTag(..), FontString(..),
    FontBlock(..), psStr, newPath, moveto, lineto,
    closepath, stroke, show_, arc, gsave, grestore,
    translate, MakeFontTags(..), wrapWithinWidth,
    box
) where
```

19.1 Data types

EPS is plain text, consisting of some header comments followed by drawing commands in PostScript. The header comments are very important as they identify the text as EPS and specify a bounding box in which the figure appears. Any drawing outside of the bounding box is clipped.

```
type EPS = String
```

A **PS** is a sequence lines of PostScript code *in reverse order*. We build up a figure in reverse order initially to avoid a lot of use of ++.

```
type PS = [String]
```

A **BPS** is a figure in construction with its PS code and a list of bounding boxes that enclose the elements of the figure.

```
type BPS = ([Box Double], PS)
```

19.2 Finalizing to EPS

bpsToEps *b* finalizes a BPS figure by reversing it and constructing the EPS header comment including the bounding box.

```
bpsToEps :: BPS -> EPS
```

19.3 Drawing in BPS

epsDraw *options x* renders *x* as a BPS, where *x* has a data type which is an instance of `EPSDrawable` and *options* contains settings that might affect the rendering.

```
class EPSDrawable a where
```

```
  epsDraw :: Options -> a -> BPS
```

19.4 Merging BPS components

joinBPS *a b Δ_x Δ_y* puts figure *b* over figure *a* displaced by Δ_x and Δ_y .

```
joinBPS :: BPS -> BPS -> Double -> Double -> BPS
```

19.5 Drawing text

19.5.1 Switching fonts efficiently

setUpFonts is PS code to find the fonts and define procedures for switching to them.

```
setUpFonts :: PS
```

19.5.2 Font metrics

times10Width *c* returns the width of a character *c* in Times-Roman 10 point, in 72 dpi pixels. (not exhaustive)

```
times10Width :: Char -> Double
```

times10ItalicWidth *c* returns the width of a character *c* in Times-Italic 10 point, in 72 dpi pixels. (not exhaustive)

```
times10ItalicWidth :: Char -> Double
```

helvetica10Width *c* returns the width of a character *c* in Helvetica 10 point, in 72 dpi pixels. (not exhaustive)

```
helvetica10Width :: Char -> Double
```

helvetica10ObliqueWidth *c* returns the width of a character *c* in Helvetica 10 point oblique, in 72 dpi pixels. (not exhaustive)

```
helvetica10ObliqueWidth :: Char -> Double
```

helvetica10BoldWidth *c* returns the width of a character *c* in Helvetica 10 point bold, in 72 dpi pixels. (not exhaustive)

```
helvetica10BoldWidth :: Char -> Double
```

helvetica10BoldObliqueWidth *c* returns the width of a character *c* in Helvetica 10 point bold oblique, in 72 dpi pixels. (not exhaustive)

```
helvetica10BoldObliqueWidth :: Char -> Double
```

symbol10Width *c* returns the width of a character *c* in Symbol 10 point, in 72 dpi pixels. (not exhaustive)

```
symbol10Width :: Char -> Double
```

19.5.3 Font tags

Type **FontTag** tags a string with either:

- **Space** – a space between symbols at which lines may be broken.
- **Times10** – Times font, roman face, 10 point;
- **Times10Italic** – Times font, italic face, 10 point;
- **Helvetica10** – Helvetica font, 10 point;
- **Helvetica10Oblique** – Helvetica font, oblique face, 10 point;
- **Helvetica10Bold** – Helvetica font, bold face, 10 point;
- **Helvetica10BoldOblique** – Helvetica font, bold-oblique face, 10 point; or
- **Symbol10** – Symbol font, 10 point.

```
data FontTag = Space
             | Times10           String
             | Times10Italic     String
             | Helvetica10      String
             | Helvetica10Oblique String
             | Helvetica10Bold  String
             | Helvetica10BoldOblique String
             | Symbol10         String
```

```
deriving Show
```

ftWidth *f* returns the total width of `FontTag f`.

```
ftWidth :: FontTag -> Double
```

19.5.4 Font strings

Type **FontString** is a sequence of Strings, tagged by the font they are to be rendering in.

```
data FontString = FontString [FontTag]
deriving Show
```

fsWidth *f* returns the total width of `FontString f`.

```
fsWidth :: FontString -> Double
```

f +++ f' catenates `FontStrings f` and `f'`, with a space added at the join.

```
(+++) :: FontString -> FontString -> FontString
```

fsWords *f* groups a `FontString` into the Space-separated `FontStrings`.

```
fsWords :: FontString -> [FontString]
```

Instances of class **MakeFontTags** may be encoded as `FontStrings`.

```
class MakeFontTags a where
```

```
  makeFontTags x renders x as a FontString
```

```
  makeFontTags :: a -> [FontTag]
```

makeFontTagsPrec *p x* renders *x* as a `FontString`, in parentheses if *p* is greater than then precedence of object *x* (as per `showsPrec`).

```
  makeFontTagsPrec :: Int -> a -> [FontTag]
```

19.5.5 Font blocks

Type **FontBlock** is a sequence of `FontStrings` to be drawn as a block.

```
data FontBlock = FontBlock [FontString]
deriving Show
```

wrapWithinWidth *w f* wraps `FontString f` at the Spaces it contains to within maximum width *w* if possible, returning the wrapped `FontBlock`.

```
wrapWithinWidth :: Double -> FontString -> FontBlock
```

wrapToAspect *f a* wraps `FontString f` at the Spaces it contains to as close to the desired aspect ratio *a* as possible, where an aspect ratio is the height divided by the width, returning the wrapped `FontBlock`.

```
wrapToAspect :: FontString -> Double -> FontBlock
```

19.5.6 Text encoding

`psStr` `cs` encodes a string for inclusion in PostScript as a literal.

```
psStr :: String -> String
```

19.6 Conveniences

Some PostScript operators: `newpath`; `moveto`; `lineto`; `closepath`; `stroke`; `show_`; `arc`; `gsave`; `grestore`; `translate`.

```
newpath = "newpath"
moveto  = "moveto"
lineto  = "lineto"
closepath = "closepath"
stroke  = "stroke"
show_   = "show"
arc     = "arc"
gsave   = "gsave"
grestore = "grestore"
translate = "translate"
```

Draw a `box`.

```
box :: Double -> Double -> Double -> Double -> BPS
```

19.7 Instance declarations

19.7.1 EPSDrawable

```
instance EPSDrawable FontTag where
```

```
instance EPSDrawable FontString where
```

```
instance EPSDrawable FontBlock where
```

20 Geometry

Module `ABR.Geometry` implements some basic geometric calculations.

```
module ABR.Geometry (
  Point, Box, Angle,
  GeoNum(
    netBox, shiftBoxes, leastRightShift,
    placeAroundOval, iGeo, iPoint, iBox,
    insetBox
  )
) where
```

20.1 Data types

A `Point` on the plane in Cartesian coordinates (x, y) . It is assumed that the coordinate system is conventional, with the y -axis the right way up, unlike most screen graphics coordinate systems. The actual numeric type is not specified, and where possible functions will be written to accommodate both any of `Float`, `Double`, `Int`, or `Integer` or `Rational`. See class `GeoNum`, below.

```
type Point a = (a, a)
```

A `Box` (l, b, r, t) is a rectangle defined by its left l , bottom b , right r and top t . It is assumed that $l \leq r$ and $b \leq t$.

```
type Box a = (a, a, a, a)
```

`Angle`s are represented in degrees. Absolute angles are measured anticlockwise from the positive x -axis.

```
type Angle a = a
```

`Line`s are represented by the coefficients of the general formula for a line (A, B, C) in:

$$Ax + by + C = 0$$

```
type Line a = (a, a, a)
```

`LineSeg`ments are represented by the start and end points.

```
type LineSeg a = (Point a, Point a)
```

20.2 Geometric computations

`GeoNum` overloads functions which perform geometric computations.

```
class (Ord a, Num a) => GeoNum a where
```

`netBox` `boxes` returns the smallest box that encloses all of `boxes`.

```
netBox :: [Box a] -> Box a
```

`shiftBoxes` `boxes` Δ_x Δ_y returns the `boxes` displaced by Δ_x and Δ_y .

```
shiftBoxes :: [Box a] -> a -> a -> [Box a]
```

`leastRightShift` `as` `bs` returns the least horizontal displacement so that list of boxes `bs` no longer overlap list of boxes `as`, as in figure 1.

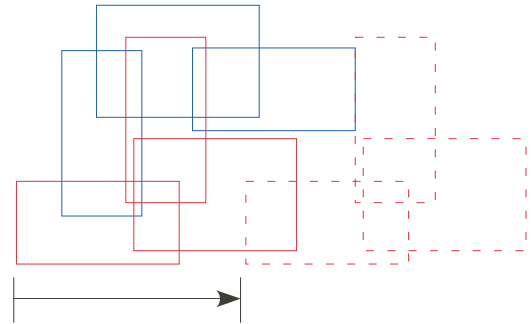


Figure 1: The least right shift to stop two sets of boxes overlapping.

```
leastRightShift :: [Box a] -> [Box a] -> a
```

`distAroundOval` n o ϕ returns a list of n points distributed around the oval inscribed in box o . The points will be equally distributed by angle of separation wrt to the centre of the oval, clockwise, and starting from angle ϕ .

```
placeAroundOval :: Int -> Box a -> Angle a -> [Point a]
```

`iGeo` x converts a `GeoNum` x to an `Int`.

```
iGeo :: a -> Int
```

`iPoint` p converts the coordinates of point p to `Ints`.

```
iPoint :: Point a -> Point Int
```

`iBox` b converts the coordinates of box b to `Ints`.

```
iBox :: Box a -> Box Int
```

`insetBox` d (l, b, r, t) reduces box (l, b, r, t) all around by distance d . It is assumed that $2d \leq r - l$ and $2d \leq t - b$, that is that the original box was big enough to do this.

```
insetBox :: a -> Box a -> Box a
```

`segToLine` $((x_1, y_1), (x_2, y_2))$ computes (A, B, C) for segment $((x_1, y_1), (x_2, y_2))$. Working:

$$A = y_2 - y_1, B = x_2 - x_1, C = -(Ax_1 + By_1)$$

```
segToLine :: LineSeg a -> Line a
```

`insetSeg` d $((x_1, y_1), (x_2, y_2))$ clips a distance d from both ends of the line segment. It is assumed that the line segment is long enough to do this.

```
insetSeg :: a -> LineSeg a -> LineSeg a
```

20.3 Instance declarations

```
instance GeoNum Float where
```

```
instance GeoNum Double where
```

```
instance GeoNum Int where
```

```
instance GeoNum Integer where
```

```
instance GeoNum Rational where
```

21 Graphs

Module `ABR.Graphs` implements a directed, unweighted graph as an ADT. This is *UNDER CONSTRUCTION*. This implementation closely follows that described by Rabhi and Lapalme [1] and Launchbury [2].

```
module ABR.Graph (
  Graph(..), SGraph, mapG, transposeG
  -- isReachable, reachable, isCyclic
) where
```

21.1 Graph abstract data type

A *vertex* is a value from some enumerated type and for implementation reasons must usually be an instance of classes `Eq`, `Ord`, `Ix` and `Show`.

An *edge* is an ordered association two vertices. Note that we are only dealing with unweighted graphs here.

```
type Edge v = (v,v)
```

A graph $G = (V, E)$ consists of a set of edges E that connect a set of vertices V . The set of edges E is a relation on V . If G is a directed graph, E is not symmetric. The number of vertices is $|V|$ and the number of edges is $|E|$.

A graph, as an abstract data type is defined by the methods of this type class.

```
class Graph g where
```

`mkGraph v v' E` builds a graph (V, E) . The set of vertices V assumed to exist is the range $[v..v']$. E is the relation defining the edges.

```
mkGraph :: Ix v => v -> v' -> [Edge v] -> g v
```

`vertices G` returns the list of vertices V in graph G .

```
vertices :: Ix v => g v -> [v]
```

`boundsG G` returns the least and greatest of the vertices V in graph G .

```
boundsG :: Ix v => g v -> (v,v)
```

`edges G` returns the list of edges E in graph G .

```
edges :: Ix v => g v -> [Edge v]
```

`adjacent G v` returns the list of vertices in graph G that can be reached from vertex v in one step.

```
adjacent :: Ix v => g v -> v -> [v]
```

`isAdjacent G v v'` returns `True` iff vertex v' in graph G can be reached from vertex v in one step.

```
isAdjacent :: Ix v => g v -> v -> v' -> Bool
```

21.2 Sparse graph type

There are several ways to represent graphs as a Haskell data structure. The following is likely to be suitable when the graph is sparse.

```
newtype SGraph vertex =
  deriving (Show)
```

```
instance Graph SGraph where
```

21.3 Graph Operations

`mapG f v v' G` builds a new graph G' formed by applying f to every edge in G , such that:

1. $G = (V, E)$
2. $G' = (V', E')$
3. $V' = [v..v']$
4. $E' = \{f e | e \in E\}$

```
mapG :: (Ix v, Ix v', Graph g, Graph g') =>
  (Edge v -> Edge v') -> v' -> v' -> g v -> g' v'
```

`transposeG G` reverses all the edges in G .

```
transposeG :: (Ix v, Graph g) => g v -> g v
```

21.4 Reachability

`isReachable g v v'` returns `True` iff a vertex v' in graph g is reachable from vertex v . A depth-first search is used. This implementation uses a mutable array so that already visited nodes can be skipped in constant time.

```
| isReachable :: (Ix v) => Graph v -> v -> v' -> Bool
```

`reachable g v` returns the list of vertices in graph g that are reachable from vertex v . A depth-first search is used. This implementation uses a mutable array so that already visited nodes can be skipped in constant time.

```
| reachable :: (Ix v) => Graph v -> v -> [v]
```

21.5 Cycles detection

`isCyclic g` returns `True` iff graph g is cyclic. A depth-first search is used.

```
| isCyclic :: (Ix v, Enum v) => Graph v -> Bool
```

22 HaskellLexer

The module `ABR.HaskellLexer` provides facilities to partially parse Haskell sources.

```
module ABR.HaskellLexer (
  deliterate, programL, offside, unlex, promoteMethods,
  discardInners, moduleName, declared, declarations
) where
```

22.1 Handling literate scripts

`deliterate cps` removes all informal text from cps , a literate Haskell source as a list of character-position pairs as produced by `Parser.preLex`. A similar list of character-position pairs is returned. This does not remove `--` or `{- -}` comments from within the formal text. Those comments are handled by the lexer.

```
deliterate :: [(Char, Pos)] -> [(Char, Pos)]
```

22.2 Lexing scripts

This section implements a lexer for Haskell. It is essentially complete for ASCII sources, but not for unicode sources.

`programL` performs the lexical analysis of any Haskell source. Apply `deliterate` to literate sources *before* lexing.

```
programL :: Lexer
```

22.3 Handling the offside rule

`offside tpls` applies the off-side layout rule, inserting braces and semicolons. $tpls$ is a list of tag-lexeme-position tuples produced by the lexer (`programL`) and after all whitespace has been removed with `dropwhite`. Note that scripts either start with an explicit or implicit module header. Either case is properly handled, as is the case of unexpectedly short scripts.

```
offside :: TLPs -> TLPs
```

22.4 Diagnostics

`unlex tpls` undoes all of the above good work by unrolling all of the lexing of $tpls$. It should be very useful to check for instance that the offside rule has been applied properly.

```
unlex :: TLPs -> String
```

22.5 Poor man's parsing

This section contains functions for analysing the results of the lexing phases above without using a real (combinator) `Parser`. This method might turn out to be good enough to generate the sort of information required to create the Haskell dictionary which started me down this path. I have also used it for logical line counting. Call this function BEFORE the next one. `promoteMethods tpls` promotes the definitions within the where clause of class declarations to the top level.

```
promoteMethods :: TLPs -> TLPs
```

This function makes it easier to pick out top-level declarations. `discardInners` *tlps* filters out all less-than-top-level declarations, crudely by eliminating all `{stuff; stuff; ... ; stuff}` sequences inside the top-level such sequence in *tlps*.

```
discardInners :: TLPs -> TLPs
```

`discard` *tlps* discards all lexemes from a TLP list, *tlps*, up to but not including the next semi-colon or opening or closing brace.

```
discard :: TLPs -> TLPs
```

`moduleName` *tlps* extracts the name of the module from a TLP list, *tlps*, if there is one, or returns `[]` if there is none.

```
moduleName :: TLPs -> TLPs
```

`declarations` *tlps* a TLP list, *tlps*, up into its top level declarations.

```
declarations :: TLPs -> [TLPs]
```

`declared` *tlps* takes a top-level declaration *tlps* and returns the type of declaration (as a new set of Tags), the names of the declared objects (Lexemes) and the positions of the names of the objects (Poss).

```
declared :: TLPs -> TLPs
```

23 LockFile

The `ABR.LockFile` module provides a facility to lock a file so that multiple concurrent processes don't destructively interfere.

```
module ABR.LockFile (  
    lockFile, unlockFile, isLockedFile, areAnyLocked,  
    lockFiles, unlockFiles, lockGuard, blockGuard  
    ) where
```

23.1 Basic lock operations

`lockFile` *path* locks the file at *path*, returning `True` iff the file was not already locked and was successfully locked. `unlockFile` *path* unlocks the file at *path*, returning `True` iff the file was locked and was successfully unlocked. `isLockedFile` *path* returns `True` iff the file at *path* is locked.

```
lockFile, unlockFile, isLockedFile :: String -> IO Bool
```

```
extend :: String -> String  
extend filename = filename ++ ".LOCK"
```

23.2 Multiple file operations

`areAnyLocked` *fs* returns `True` iff at least one of the files named in *fs* is locked.

```
areAnyLocked :: [String] -> IO Bool
```

`lockFiles` *fs* locks all files named in *fs*. `unlockFiles` *fs* unlocks all files named in *fs*.

```
lockFiles, unlockFiles :: [String] -> IO ()
```

23.3 Guards

`lockGuard` *directory fs handler process* checks whether any of the files *fs* in *directory* are locked. If any one is, *handler* is executed, otherwise *process* is executed.

`blockGuard` *directory fs handler process* checks whether any of the files *fs* in *directory* are locked. If any one is, *handler* is executed, otherwise the files are locked, *process* is executed, then the files are unlocked again.

```
lockGuard, blockGuard ::  
    String -> [String] -> IO () -> IO () -> IO ()
```

24 MySQL C API Binding

Module `ABR.MySQLCBinding` is an interface to the MySQL C API. It uses C types for all arguments and results, without any attempt to make it Haskell friendly. Module `MySQL`, which is built on top of this module, provides an interface using Haskell types.

The descriptions have been adapted from the MySQL Reference Manual, omitting much detail, and introducing new errors. I'd have that close at hand while using this module.

```
module ABR.MySQLCBinding (  
    My_bool, My_ulonglong, MYSQL, MYSQL_RES,  
    MYSQL_ROW, MYSQL_ROW_OFFSET, MYSQL_FIELD,  
    MYSQL_FIELD_OFFSET, Enum_mysql_option,  
    mysql_affected_rows, mysql_change_user,  
    mysql_character_set_name, mysql_close,  
    mysql_data_seek, mysql_errno, mysql_error,  
    mysql_fetch_field, mysql_fetch_fields,  
    mysql_fetch_field_direct, mysql_fetch_lengths,  
    mysql_fetch_row, mysql_field_count,  
    mysql_field_seek, mysql_field_tell,  
    mysql_free_result, mysql_get_client_info,  
    mysql_get_host_info, mysql_get_proto_info,  
    mysql_get_server_info, mysql_info, mysql_init,  
    mysql_insert_id, mysql_kill, mysql_list_dbs,  
    mysql_list_fields, mysql_list_processes,  
    mysql_list_tables, mysql_num_fields,  
    mysql_num_rows, mysql_options, mysql_ping,  
    mysql_query, mysql_real_connect,  
    mysql_real_escape_string, mysql_real_query,  
    mysql_row_seek, mysql_row_tell, mysql_select_db,  
    mysql_shutdown, mysql_stat, mysql_store_result,  
    mysql_thread_id, mysql_use_result  
    ) where
```

24.1 API Data types

24.1.1 Basics

A `My_bool` is a C char.

```
type My_bool = CChar
```

A `My_ulonglong` is supposedly a 64 bit, unsigned integer, but the way it is used implies signed is a more useful choice.

```
type My_ulonglong = CLLong
```

24.1.2 Connections

A `MYSQL` is some opaque C object. A pointer to this structure is our handle on a connection.

```
type MYSQL =
```

24.1.3 Results

A `MYSQL_RES` is some opaque C object. A pointer to this structure is our handle on a result to a query.

```
type MYSQL_RES =
```

24.1.4 Rows

A `MYSQL_ROW` is an array of strings. The strings are *not* terminated with `\0`s, as they could be binary data.

```
type MYSQL_ROW = Ptr CString
```

A `MYSQL_ROW_OFFSET` is a pointer to a `MYSQL_ROWS`.

```
type MYSQL_ROW_OFFSET =
```

24.1.5 Fields

A `MYSQL_FIELD` is a C structure.

```
type MYSQL_FIELD =
```

A `MYSQL_FIELD_OFFSET` is an offset into a MySQL field list.

```
type MYSQL_FIELD_OFFSET = CUInt
```

24.1.6 Options

An `Enum_mysql_option` is a C enumeration, used by function `mysql_options`.

```
type Enum_mysql_option = CUInt
```

24.2 API Functions

mysql_affected_rows *mysql* returns the number of rows changed by the last UPDATE, deleted by the last DELETE or inserted by the last INSERT statement. May be called immediately after `mysql_query` for UPDATE, DELETE, or INSERT statements. For SELECT statements, `mysql_affected_rows` works like `mysql_num_rows`. Returns an integer greater than zero to indicate the number of rows affected or retrieved. Zero indicates that no records were updated for an UPDATE statement, no rows matched the WHERE clause in the query or that no query has yet been executed. -1 indicates that the query returned an error or that, for a SELECT query, `mysql_affected_rows` was called prior to calling `mysql_store_result`.

```
mysql_affected_rows :: Ptr MYSQL -> IO My_ulonglong
```

mysql_change_user *mysql user passwd db* changes the user to *user* with *passwd* and causes the database specified by *db* to become the default (current) database on the connection specified by *mysql*. In subsequent queries, this database is the default for table references that do not include an explicit database specifier. Returns zero for success, non-zero if an error occurred.

```
mysql_change_user :: Ptr MYSQL -> CString -> CString  
-> CString -> IO My_bool
```

mysql_character_set_name *mysql* returns the default character set for the current connection.

```
mysql_character_set_name :: Ptr MYSQL -> IO CString
```

mysql_close *mysql* closes and deallocates the connection *mysql*.

```
mysql_close :: Ptr MYSQL -> IO ()
```

mysql_data_seek *result offset* seeks to an arbitrary row in a query result set. This requires that the result set structure contains the entire result of the query, so `mysql_data_seek` may be used in conjunction only with `mysql_store_result`, not with `mysql_use_result`. The offset should be a value in the range from 0 to `mysql_num_rows result - 1`.

```
mysql_data_seek :: Ptr MYSQL_RES -> CULong -> IO ()
```

mysql_errno *mysql* returns the error code returned by the last MySQL API function. 0 indicates no error.

```
mysql_errno :: Ptr MYSQL -> IO CUInt
```

mysql_error *mysql* returns the error message returned by the last MySQL API function. An empty string indicates no error.

```
mysql_error :: Ptr MYSQL -> IO CString
```

mysql_fetch_field *result* returns the definition of one column of a result set as a `MYSQL_FIELD` structure. Call this function repeatedly to retrieve information about all columns in the result set. Returns NULL when no more fields are left. `mysql_fetch_field` is reset to return information about the first field each time you execute a new SELECT query. The field returned is also affected by calls to `mysql_field_seek`.

```
mysql_fetch_field :: Ptr MYSQL_RES  
-> IO (Ptr MYSQL_FIELD)
```

mysql_fetch_fields *result* returns an array of all `MYSQL_FIELD` structures for a result set. Each structure provides the field definition for one column of the result set.

```
mysql_fetch_fields :: Ptr MYSQL_RES  
-> IO (Ptr MYSQL_FIELD)
```

mysql_fetch_field_direct *result fieldnr* returns column *fieldnr*'s field definition as a `MYSQL_FIELD` structure. The value of *fieldnr* should be in the range from 0 to `mysql_num_fields result - 1`.

```
mysql_fetch_field_direct :: Ptr MYSQL_RES -> CUInt  
-> IO (Ptr MYSQL_FIELD)
```

mysql_fetch_lengths *result* returns an array of the lengths of the columns of the current row within a result set, or NULL in the case of an error.

```
mysql_fetch_lengths :: Ptr MYSQL_RES -> IO (Ptr CULong)
```

mysql_fetch_row *result* retrieves the next row of a result set. Returns NULL when there are no more rows to retrieve.

```
mysql_fetch_row :: Ptr MYSQL_RES -> IO (MYSQL_ROW)
```

mysql_field_count *mysql* returns the number of columns for the most recent query on the connection.

```
mysql_field_count :: Ptr MYSQL -> IO CUInt
```

mysql_field_seek *result offset* sets the field cursor to the given offset. The next call to `mysql_fetch_field` will retrieve the field definition of the column associated with that offset. To seek to the beginning of a row, pass an offset value of zero. Returns the previous value of the field cursor.

```
mysql_field_seek :: Ptr MYSQL_RES  
-> MYSQL_FIELD_OFFSET -> IO MYSQL_FIELD_OFFSET
```

mysql_field_tell *result* returns the position of the field cursor used for the last `mysql_fetch_field`. This value can be used as an argument to `mysql_field_seek`.

```
mysql_field_tell :: Ptr MYSQL_RES  
-> IO MYSQL_FIELD_OFFSET
```

mysql_free_result *result* frees the memory allocated for a result set by `mysql_store_result`, `mysql_use_result`, `mysql_list_dbs`, etc. When you are done with a result set, you must free the memory it uses by calling this function.

```
mysql_free_result :: Ptr MYSQL_RES -> IO ()
```

mysql_get_client_info returns a string that represents the client library version.

```
mysql_get_client_info :: IO CString
```

mysql_get_host_info *mysql* returns a string describing the type of connection in use, including the server host name.

```
mysql_get_host_info :: Ptr MYSQL -> IO CString
```

mysql_get_proto_info *mysql* returns the protocol version used by current connection.

```
mysql_get_proto_info :: Ptr MYSQL -> IO CUInt
```

mysql_get_server_info *mysql* returns a string describing the type of connection in use, including the server host name.

```
mysql_get_server_info :: Ptr MYSQL -> IO CString
```

mysql_info *mysql* retrieves a string providing information about the most recently executed query, but only for some statements. For other statements, `mysql_info` returns NULL.

```
mysql_info :: Ptr MYSQL -> IO CString
```

mysql_init *mysql* allocates or initializes a MySQL object suitable for `mysql_real_connect`. If *mysql* is a null pointer (use `Foreign.nullPtr`), the function allocates, initializes and returns a new object. Otherwise the object is initialized and the address returned.

```
mysql_init :: Ptr MYSQL -> IO (Ptr MYSQL)
```

mysql_insert_id *mysql* returns the ID generated for an `AUTO_INCREMENT` column by the previous query. Use this function after you have performed an INSERT query into a table that contains an `AUTO_INCREMENT` field.

```
mysql_insert_id :: Ptr MYSQL -> IO My_ulonglong
```

mysql_kill *mysql pid* asks the server to kill the thread specified by *pid*.

```
mysql_kill :: Ptr MYSQL -> CULong -> IO CInt
```

mysql_list_dbs *mysql wild* returns a result set consisting of database names on the server that match the simple regular expression specified by the *wild* parameter. *wild* may contain the wild-card characters '%' or '_', or may be a NULL pointer to match all databases. Returns NULL if an error occurred.

```
mysql_list_dbs :: Ptr MYSQL -> CString
-> IO (Ptr MYSQL_RES)
```

mysql_list_fields *mysql table wild* returns a result set consisting of field names in the given table that match the simple regular expression specified by the *wild* parameter. *wild* may contain the wild-card characters '%' or '_', or may be a NULL pointer to match all fields. Returns NULL if an error occurred.

```
mysql_list_fields :: Ptr MYSQL -> CString -> CString
-> IO (Ptr MYSQL_RES)
```

mysql_list_processes *mysql* returns a result set describing the current server threads. Returns NULL if an error occurred.

```
mysql_list_processes :: Ptr MYSQL -> IO (Ptr MYSQL_RES)
```

mysql_list_tables *mysql wild* returns a result set consisting of table names in the current database that match the simple regular expression specified by the *wild* parameter. *wild* may contain the wild-card characters '%' or '_', or may be a NULL pointer to match all databases. Returns NULL if an error occurred.

```
mysql_list_tables :: Ptr MYSQL -> CString
-> IO (Ptr MYSQL_RES)
```

mysql_num_fields *result* returns the number of columns in a result set.

```
mysql_num_fields :: Ptr MYSQL_RES -> IO CUInt
```

mysql_num_rows *result* returns the number of rows in the result set.

```
mysql_num_rows :: Ptr MYSQL_RES -> IO My_ulonglong
```

mysql_options *mysql option arg* Can be used to set extra connect options and affect behavior for a connection.

```
mysql_options :: Ptr MYSQL -> Enum_mysql_option
-> CString -> IO CInt
```

mysql_ping *mysql* checks whether or not the connection to the server is working. If it has gone down, an automatic reconnection is attempted. This function can be used by clients that remain idle for a long while, to check whether or not the server has closed the connection and reconnect if necessary. Returns zero if the server is alive, non-zero otherwise.

```
mysql_ping :: Ptr MYSQL -> IO CInt
```

mysql_query *mysql query* executes the SQL query pointed to by the null-terminated string *query*. The query must consist of a single SQL statement. You should not add a terminating semicolon (;) or \g to the statement. *mysql_query* cannot be used for queries that contain binary data; you should use *mysql_real_query* instead. (Binary data may contain the '\0' character.)

```
mysql_query :: Ptr MYSQL -> CString -> IO CInt
```

mysql_real_connect *mysql host user passwd db port unix_socket client_flag* attempts to establish a connection to a MySQL database engine running on *host*. *mysql_real_connect* must complete successfully before you can execute nearly all of the other API functions. The function returns *mysql* if successful, otherwise null. The parameters are specified as follows:

mysql is a pointer to an existing MYSQL structure, initialized by *mysql_init*.

host may be either a hostname or an IP address. If null or "localhost" the local host is assumed.

user is the MySQL login ID. If null, the current user is assumed.

passwd is the password for *user* (unencrypted). If null, only users that have empty passwords are checked.

db is the database name. If not null, the connection will set the default database to this value.

port If not 0, sets the port number to use.

unix_socket If not null, sets the socket or named pipe to use.

client_flag usually 0, but can be used to cope with some special circumstances.

```
mysql_real_connect :: Ptr MYSQL -> CString -> CString
-> CString -> CString -> CUInt -> CString -> CUInt
-> IO (Ptr MYSQL)
```

mysql_real_escape_string *mysql to from length* creates a legal SQL string that you can use in a SQL statement. The string in *from* is encoded to an escaped SQL string, taking into account the current character set of the connection. The result is placed in *to* and a terminating null byte is appended. Characters encoded are NUL (ASCII 0), '\n', '\r', '\', ',', "'", and Control-Z.

```
mysql_real_escape_string :: Ptr MYSQL -> CString
-> CString -> CUInt -> IO (CInt)
```

mysql_real_query *mysql query length* executes the SQL query pointed to by *query*, which should be a string *length* bytes long. The *query* must consist of a single SQL statement. You should not add a terminating semicolon (;) or \g to the statement. You must use *mysql_real_query* rather than *mysql_query* for queries that contain binary data, because binary data may contain the '\0' character. In addition, *mysql_real_query* is faster than *mysql_query* because it does not call *strlen()* on the *query* string. If you want to know if the query should return a result set or not, you can use *mysql_field_count* to check for this. Returns zero if the query was successful, or non-zero if an error occurred.

```
mysql_real_query :: Ptr MYSQL -> CString -> CUInt
-> IO (CInt)
```

mysql_row_seek *result offset* sets the row cursor to an arbitrary row in a query result set. This requires that the result set structure contains the entire result of the query, so *mysql_row_seek* may be used in conjunction only with *mysql_store_result*, not with *mysql_use_result*. The offset should be a value returned from a call to *mysql_row_tell* or to *mysql_row_seek*. This value is not simply a row number; if you want to seek to a row within a result set using a row number, use *mysql_data_seek* instead. Returns the previous value of the row cursor. This value may be passed to a subsequent call to *mysql_row_seek*.

```
mysql_row_seek :: Ptr MYSQL_RES -> MYSQL_ROW_OFFSET
-> IO MYSQL_ROW_OFFSET
```

mysql_row_tell *result* returns the current position of the row cursor for the last *mysql_fetch_row*. This value can be used as an argument to *mysql_row_seek*. You should use *mysql_row_tell* only after *mysql_store_result*, not after *mysql_use_result*.

```
mysql_row_tell :: Ptr MYSQL_RES -> IO MYSQL_ROW_OFFSET
```

mysql_select_db *mysql db* causes the database specified by *db* to become the default (current) database on the connection specified by *mysql*. Returns zero for success or non-zero otherwise.

```
mysql_select_db :: Ptr MYSQL -> CString -> IO CInt
```

mysql_shutdown *mysql level* asks the database server to shut down. The connected user must have shutdown privileges.

```
mysql_shutdown :: Ptr MYSQL -> CInt -> IO CInt
```

mysql_stat *mysql* returns a character string containing information similar to that provided by the *mysqladmin* status command. This includes uptime in seconds and the number of running threads, questions, reloads, and open tables, or NULL if an error occurred.

```
mysql_stat :: Ptr MYSQL -> IO CString
```

mysql_store_result *mysql* reads and returns a pointer to the entire result for the last query, or NULL if there has been an error.

```
mysql_store_result :: Ptr MYSQL
-> IO (Ptr MYSQL_RES)
```

mysql_thread_id *mysql* returns the thread ID of the current connection. This value can be used as an argument to *mysql_kill* to kill the thread. If the connection is lost and you reconnect with *mysql_ping*, the thread ID will change. This means you should not get the thread ID and store it for later. You should get it when you need it.

```
mysql_thread_id :: Ptr MYSQL -> IO CULong
```

`mysql_use_result` *mysql* initiates a result set retrieval but does not actually read the result set into the client like `mysql_store_result` does. Instead, each row must be retrieved individually by making calls to `mysql_fetch_row`.

```
mysql_use_result :: Ptr MYSQL -> IO (Ptr MYSQL_RES)
```

25 MySQL Haskell API

Module `ABR.MySQL` is a Haskell interface to MySQL. This interface presents only Haskell data types, and restricts or hides many options provided by the C API.

```
module ABR.MySQL (
    MySQL, myConnect, myClose, myQuery, myFetch
) where
```

25.1 Data types

25.1.1 Connections

A `MySQL` is our handle on a MySQL connection.

```
type MySQL =
```

25.2 Functions

25.2.1 Establishing a connection

`myConnect` *host user passwd db* returns `CheckPass mysql` if a connection could be established to the MySQL server running on *host* ("*host*" = "localhost"), as *user* ("*user*" = the current user), with password *passwd* ("*passwd*" = no password), using database *db* ("*db*" = no database selected). If an error occurs, `CheckFail (errNum, errMsg)` is returned.

```
myConnect :: String -> String -> String -> String
           -> IO (CheckResult MySQL (Integer, String))
```

25.2.2 Closing a connection

`myClose` *mysql* closes the connection and frees the memory it uses.

```
myClose :: MySQL -> IO ()
```

25.2.3 Issuing a query

`myQuery` *mysql query* executes the SQL *query*, returning `CheckPass (fields, rows)` if successful, where *fields* is the number of fields that would be in any result set fetched after this query and *rows* is the number of rows affected by this query or `-1` if there is a result to be fetched, or `CheckFail (errNum, errMsg)` if not.

```
myQuery :: MySQL -> String
         -> IO (CheckResult (Int,Integer) (Integer, String))
```

25.2.4 Fetching query results

`myFetch` *mysql fields* fetches the results set for the last query. *fields* is the number of fields that will be returned, as reported by the last call to `myQuery`. It returns `CheckPass (rows, lss, csss)`, where *rows* is the number of rows in the data set, *lss* is the list of lengths of each field for each row, and *csss* is the list of rows of columns. In the case of an error, `CheckFail (errNum, errMsg)` is returned instead.

```
myFetch :: MySQL -> Int
         -> IO (CheckResult (Integer, [[Int]],
                             [[String]]) (Integer, String))
```

This has not been tested the case of NULL field values, where the row contains a null pointer.

26 Parser.Pos

The `ABR.Parser.Pos` defines a type for describing the position in a source code.

```
module ABR.Parser.Pos (
    Line, Col, Pos, HasPos(..), precedes
) where
```

26.1 Positions in a source

To report error the position, `Pos`, of a character or token in a source is required. The first line and column are indicated with `Line` and `Col` values of 0. A negative `Line` value indicates "Don't know where".

```
type Line = Int
type Col  = Int
type Pos  = (Line, Col)
```

26.2 Overloaded projector

Positions get embedded in all kinds of parse tree types. Having one overloaded function that projects out a `Pos` is useful. Make parse tree types with positions in them an instance of `HasPos`.

```
class HasPos a where
```

```
getPos x returns the position of x. pos is a legacy alias.
getPos, pos :: a -> Pos
getPos = error "undefined HasPos instance"
pos = getPos
```

26.2.1 Container instances

```
instance (HasPos a, HasPos b) => HasPos (Either a b) where
```

26.3 Relative positions

p1 `precedes` *p2* if *p1* comes earlier in than *p2*.

```
precedes :: Pos -> Pos -> Bool
```

27 Parsing

The `ABR.Parser` module provides a framework for lexical analysis and parsing using parser combinators [3, 4].

```
module ABR.Parser (
    Msg, Could(Fail, Error, OK), Analyser, succeedA,
    epsilonA, failA, errorA, (<|>), (<*>), (@>), (#>),
    cons, some, many, optional, someUntil, manyUntil, (*>), (<*>),
    alsoSat, alsoNotSat, dataSatisfies, dataSatisfies', total, nofail,
    nofail', preLex, Lexeme, Tag, Lexer, TLP, TLPs, satisfyL,
    literalL, (%>), (<***>), (<+>), (%>), soft, tagFilter,
    tokenL, endl, listL, Parser, tagP, lineNo, literalP, errMsg, warnMsg
) where
```

27.1 Error messages

An error message, `Msg`, generated by an analyser is a `String`.

```
type Msg = String
```

27.2 Results

An analyser could succeed, fail or generate an error. The `Could` type wraps around any other type to indicate success (with `OK`), failure (with `Fail`), or an immediately identifiable error (with `Error`). Failure or error values return a diagnostic message, and a position in the source. Failure means: "It's not that, try something else". An error is unrecoverable.

```
data Could a = Fail Pos Msg | Error Pos Msg | OK a
             deriving (Eq, Ord, Show)
```

27.3 Analysers

An `Analyser` is a higher-level abstraction of both lexers and parsers. An analyser is a function that tries to accept a list of inputs of type *a* with their positions, and return a value constructed from consumed inputs of type *b* (a parse tree for example), and any unconsumed inputs. Alternately it could fail or generate an error. By convention, functions that are analysers have names that end with a capital A.

```
type Analyser a b = [(a,Pos)] -> Could (b, [(a,Pos)])
```

27.4 Elementary analysers

This is the simplest analyser. `succeedA` v succeeds with a predetermined value v and does not consume any input.

```
succeedA :: b -> Analyser a b
```

`epsilonA` is the trivial case of `succeedA`. It always succeeds and returns the trivial value `()`. This implements ϵ , the symbol that stands for an empty character sequence in grammars.

```
epsilonA :: Analyser a ()
```

`failA` msg always fails with a diagnostic message msg and the position of the next input returned.

```
failA :: Msg -> Analyser a b
```

`errorA` msg always returns an error with a diagnostic message msg and the position of the next input.

```
errorA :: Msg -> Analyser a b
```

`endA` succeeds if there is no input left and returns the trivial value `()`.⁴

```
endA :: Analyser a ()
```

27.5 Elementary analyser combinators

These combinators allow the composition of analysers.

`<|>` is the alternation combinator. $a_1 <|> a_2$ returns the result of a_1 , or a_2 if a_1 failed.

```
(<|>) :: Analyser a b -> Analyser a b -> Analyser a b
```

`<*>` is the sequence combinator. $a_1 <*> a_2$ returns a pair formed from the results of a_1 and a_2 .

```
(<*>) :: Analyser a b -> Analyser a c -> Analyser a (b,c)
```

27.6 Analyser result modifiers

These functions modify an analyser by modifying the type of value it returns.

`@>` f changes the value returned by analyser a by applying function f to it.

```
(>) :: Analyser a b -> (b -> c) -> Analyser a c
```

`#>` v changes the value returned by analyser a by replacing it with v .

```
(>) :: Analyser a b -> c -> Analyser a c
```

27.7 More analyser combinators

This definition of `cons` as an uncurried form of `:` is used below.

```
cons :: (a, [a]) -> [a]
```

`some` a changes analyser a which recognizes one thing into an analyser that recognizes a sequence of *one* or more things, returned in a list. `many` a changes analyser a which recognizes one thing into an analyser that recognizes a sequence of *zero* or more things, returned in a list. `optional` a changes analyser a which recognizes one thing into an analyser that recognizes either zero or one things. An empty or singleton list of things is returned.

```
some, many, optional :: Analyser a b -> Analyser a [b]
```

`someUntil` $a_1 a_2$ creates an analyser that recognizes a sequence of one or more of the things recognized by analyser a_1 , like `some`, but stops consuming input when a second analyser a_2 would also work. `manyUntil` $a_1 a_2$ creates an analyser that recognizes a sequence of zero or more of the things recognized by analyser a_1 , like `many`, but stops consuming input when a second analyser a_2 would also work.

```
someUntil, manyUntil :: Analyser a b -> Analyser a c -> Analyser a [b]
```

`*>` is the same as `<*>`, but discards the first value.

```
(>*) :: Analyser a b -> Analyser a c -> Analyser a c
```

⁴Thanks to Chris English for suggesting this strategy.

`<*>` is the same as `<*>`, but discards second value.

```
(<*>) :: Analyser a b -> Analyser a c -> Analyser a b
```

`alsoSat` $a_1 a_2$ permits analyser a_1 to succeed and consume input iff a_2 would also succeed. `alsoNotSat` $a_1 a_2$ permits analyser a_1 to succeed and consume input iff a_2 would not succeed.

```
alsoSat, alsoNotSat :: Analyser a b -> Analyser a c -> Analyser a b
```

`dataSatisfies` $a test$ permits analyser a to succeed and consume input only if an auxiliary $test$ performed on the data returned by a returns `True`.

```
dataSatisfies :: Analyser a b -> (b -> Bool) -> Analyser a b
```

`dataSatisfies'` $a test msg$ permits analyser a to succeed and consume input only if an auxiliary $test$ performed on the data returned by a returns `True`. If this test fails, a `Fail` is returned with msg .

```
dataSatisfies' :: Analyser a b -> (b -> Bool) -> Msg -> Analyser a b
```

`total` a forces analyser a to fail if it can't consume all the inputs. `total` :: `Analyser a b -> Analyser a b`

`nofail` a forces analyser a to return an error when otherwise it would merely fail.

```
nofail :: Analyser a b -> Analyser a b
```

`nofail'` $a msg$ forces analyser a to return an error when otherwise it would merely fail and overrides the error message with msg .

```
nofail' :: Msg -> Analyser a b -> Analyser a b
```

27.8 Lexers

Lexing is the process of breaking the input stream of characters up into a stream of lexemes (tokens). Before lexing can take place, the locations must be added.

`preLex` cs returns all of the characters in cs paired with its position.

```
preLex :: [Char] -> [(Char,Pos)]
```

Each `Lexeme` must be identified as belonging to one of an expected set of classes of lexemes. This information will be passed from a lexer to a parser by use of a `Tag`.

```
type Lexeme = String
type Tag     = String
```

The input to a `Lexer` function is a list of characters and their positions. The output from a lexer is a list of lexemes with their tags and positions and the list of unconsumed characters and their positions. The output could also be an error or failure message. By convention, a function that is a lexer has a name that ends with a capital L.

```
type Lexer = Analyser Char [(Tag, Lexeme), Pos]
```

Lexers produce streams of tagged lexemes. These shorthand type synonyms, `TLP` and `TLPs` are useful when writing functions that process lexed sources.

```
type TLP = ((Tag, Lexeme), Pos)
type TLPs = [TLP]
```

27.9 Elementary lexers

`satisfyL` $p tag$ succeeds if the first input character passes test p . On success the lexeme returned is the string containing just that character and the tag returned is tag . On failure the message "`tag expected.`" is returned.

```
satisfyL :: (Char -> Bool) -> Tag -> Lexer
```

`literalL` c succeeds if the first input is c . On success the lexeme returned is $[c]$ with the tag `''c''`. On failure the message `''c'' expected.` is returned.

```
literalL :: Char -> Lexer
```

These lexers are only capable of accepting single characters. To recognize and return tokens made up of multiple characters, we use the combinators described below.

27.10 Special combinators for lexers

`l %>` *tag* overrides the tag produced by lexer *l* with *tag*.

```
( %> ) :: Lexer -> Tag -> Lexer
```

`l1 <*>` *l2* “hard”-sequences two lexers *l1* and *l2*. The tag returned is the space-separated catenation of the two tags, the lexeme returned is the catenation of the two lexemes, and the position returned is the first position.

```
( <*> ) :: Lexer -> Lexer -> Lexer
```

`l1 <+>` *l2* “soft”-sequences two lexers *l1* and *l2*. The combined lexer returns the catenation of the lists of lexemes produced by each Lexer.

```
( <+> ) :: Lexer -> Lexer -> Lexer
```

`l *%>` *tag* modifies a lexer *l* by catenation of all its returned lexemes and returning the supplied *tag*. The position returned is the first. This permits the use of the combinators `some`, `many` and `optional` (above).

```
( *%> ) ::
  Analyser Char [[(Tag, Lexeme), Pos]] -> Tag -> Lexer
```

`soft` (*k l*) returns a lexer by soft catenating the result of some combinator $k \in \{\text{some, many, optional, ...}\}$ applied to a lexer *l*.

```
soft :: Analyser Char [[(Tag, Lexeme), Pos]] -> Lexer
```

`tagFilter` *tag l* modifies lexer *l* by making it throw out lexemes with a specified *tag*.

```
tagFilter :: Tag -> Lexer -> Lexer
```

27.11 Frequently used lexers

`tokenL` *token* recognizes a particular *token*.

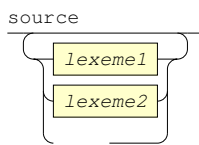
```
tokenL :: String -> Lexer
```

`endL` succeeds at the end of input, returning no lexemes.

```
endL :: Lexer
```

This is a common lexical structure for sources:

```
source ::= { $lexeme1$ | $lexeme2$ | $...$ }.
```



`listL` *ls* builds a lexer for *source* out of a list of alternative lexers *ls*.

```
listL :: [Lexer] -> Lexer
```

27.12 Parsing

Parsing is the process of transforming the stream of lexemes into a parse tree. The input to a `Parser` is the list of lexemes with their tags and positions. The output from a parser is the parse tree (of type *a* and the list of unconsumed lexemes, or failure or error. By convention, a function that is a parser has a name that ends with a capital P.

```
type Parser a = Analyser (Tag, Lexeme) a
```

27.13 Elementary parsers

`tagP` *tag* creates a parser that succeeds if the first lexeme has the supplied *tag*. On failure, the message returned is “*tag* expected.”.

```
tagP :: Tag -> Parser (Tag, Lexeme, Pos)
```

`literalP` *tag lexeme* creates a parser that succeeds if the first tag and lexeme match the supplied *tag* and *lexeme*. On failure, the message returned is “*tag* “*lexeme*” expected.”.

```
literalP :: Tag -> Lexeme -> Parser (Tag, Lexeme, Pos)
```

27.14 Parser result modifiers

`lineNo` *p* extracts just the line number. *p* is a parser with the same result type as `tagP`.

```
lineNo :: Parser (Tag, Lexeme, Pos) -> Parser Int
```

27.15 Error reporting

`errMsg` *position message source* generates an error message that reports the error *message* and the offending *position* in the *source*.

`warnMsg` does the same, but tags the output as only a warning.

```
errMsg, warnMsg :: Pos -> Msg -> String -> String
```

27.16 Instance declarations

instance HasPos (Could a) where

28 Parser.Lexers

The `ABR.Parser.Lexers` module provides some frequently used lexers for common syntactic elements.

```
module ABR.Parser.Lexers (
  spaceL, tabL, vertabL, formfeedL, newlineL,
  whitespaceL, dropWhite, stringL, cardinalL, fixedL,
  floatL, signedCardinalL, signedFixedL, signedFloatL
) where
```

28.1 Frequently used lexers

`spaceL`, `tabL`, `newlineL`, `vertabL`, `formfeedL`, and `returnL` recognize individual whitespace characters.

```
space ::= " ".
```



```
tab ::= "\\t".
```



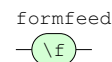
```
newline ::= "\\n".
```



```
vertab ::= "\\v".
```



```
formfeed ::= "\\f".
```



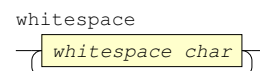
```
return ::= "\\r".
```



```
spaceL, tabL, newlineL, vertabL, formfeedL, returnL
  :: Lexer
```

`whitespaceL` recognizes any amount of whitespace, returning it with tag “ ”.

```
whitespace ::= { $whitespace char$ }+.
```



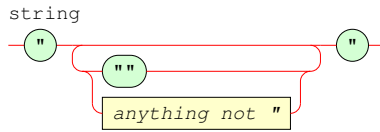
```
whitespaceL :: Lexer
```

dropWhite *l* modifies *l* by filtering out lexemes. with tag " ".

`dropWhite :: Lexer -> Lexer`

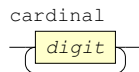
stringL recognizes strings delimited by double quotes that may extend across many lines. Use two double quotes for one, *à la Pascal*.

```
string ::= "\"" {"\"|" | $anything not \"$"} "\"";
level="lexical".
```



cardinalL recognizes a cardinal number, a sequence of decimal digits.

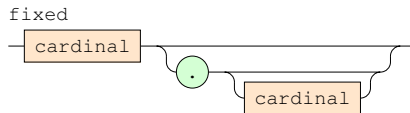
```
cardinal ::= {$digit$}+.
```



`cardinalL :: Lexer`

fixedL recognizes an unsigned fractional number with no exponent.

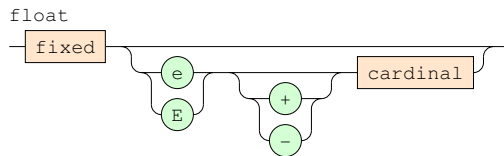
```
fixed ::= cardinal [ "." [cardinal]].
```



`fixedL :: Lexer`

floatL recognizes an unsigned floating point number.

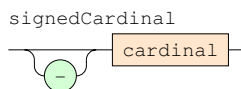
```
float ::= fixed [ ("e"|"E") ["+" | "-"] cardinal].
```



`floatL :: Lexer`

signedCardinalL recognizes a signed whole number.

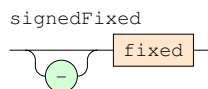
```
signedCardinal ::= ["-"] cardinal.
```



`signedCardinalL :: Lexer`

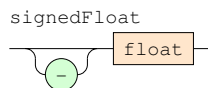
signedFixedL recognizes a signed fixed number.

```
signedFixed ::= ["-"] fixed.
```



`signedFixedL :: Lexer`

```
signedFloat ::= ["-"] float.
```



`signedFloatL :: Lexer`

29 Parser.Checks

The **ABR.Parser.Checks** module provides a some functions for easy implementation of the parsing sequence.

`module ABR.Parser.Checks (checkLex, checkParse) where`

29.1 Easy lexer and parser sequencing

checkLex *l source* uses the check abstraction to sequence the prelexing of the *source*, lexing using *l*, error detection and construction of error messages.

```
checkLex ::
  Lexer -> Check String [(Tag, Lexeme), Pos] String
```

checkParse *l p source* uses the check abstraction to sequence the prelexing of the *source*, lexing using *l*, parsing using *p*, error detection and construction of error messages.

```
checkParse :: Lexer -> Parser a -> Check String a String
```

30 Playing Cards

Module **ABR.PlayingCards** provides basic data types for card playing games and problems.

```
module ABR.PlayingCards (
  Suit(..), Rank(..), suits, ranks, Card(..), Deck,
  deck52, deck54, Hand, shuffle
) where
```

30.1 Data types

Most cards are members of one of these **Suit** s: **Clubs** (♣); **Diamonds** (♦); **Hearts** (♥); **Spades** (♠).

```
data Suit = Clubs | Diamonds | Hearts | Spades
  deriving (Eq, Ord, Enum, Bounded)
```

Most cards have of one of these **Rank** s: **Ace**; **R2**; **R3**; **R4**; **R5**; **R6**; **R7**; **R8**; **R9**; **R10**; **Jack**; **Queen**; **King**.

```
data Rank = Ace | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 |
  R10 | Jack | Queen | King
  deriving (Eq, Ord, Enum, Bounded)
```

ranks is the set of all ranks. **suits** is the set of all suits.

```
ranks :: [Rank]
suits :: [Suit]
```

A **Card** is either a card belonging to one of the **Suit** s (with a **rank** and a **suit**), or is a **Joker**.

```
data Card = Suit {rank :: Rank,
  suit :: Suit}
  | Joker
  deriving (Eq)
```

A **Deck** of cards, or a **Hand** of cards:

```
type Deck = [Card]
type Hand = [Card]
```

30.2 Creating decks

deck52 returns all the cards in a standard 52-card deck. **deck54** returns all the cards in a standard 52-card deck plus 2 jokers.

```
deck52, deck54 :: Deck
```

shuffle *deck* returns all the cards in *deck* in a new random order.

```
shuffle :: Deck -> IO Deck
```

30.3 Instance declarations

```
instance Show Suit where
```

```
instance Show Rank where
```

```
instance Show Card where
```

31 Poker

Module `ABR.Poker` provides basic stuff like categorisation of hands, not tactics.

```
module ABR.Poker (  
  sortByRankSuit, sortBySuitRank, groupByRank,  
  groupBySuit, areSuccRanks, HandType(..), handType,  
  isGarbage, isPair, isTwoPair, isTriple, isStraight,  
  isFlush, isFullHouse, isPoker, isStraightFlush,  
  compareCards, compareGroups, compareHands, beats,  
  ties  
) where
```

31.1 Presorting and grouping

`sortByRankSuit` *cs* sorts *cs* by rank and then suit.

`sortBySuitRank` *cs* sorts *cs* by suit and then rank.

`sortByRankSuit, sortBySuitRank` :: [Card] -> [Card]

`groupByRank` *cs* groups the cards in *cs* by common ranks. Each group will be sorted by suit. All the groups are sorted by the length of the group. `groupBySuit` *cs* groups the cards in *cs* by common suits. Each group will be sorted by rank. All the groups are sorted by the length of the group.

`groupByRank, groupBySuit` :: [Card] -> [[Card]]

`areSuccRanks` *r r'* returns True iff *r'* is the next highest rank after *r*.

`areSuccRanks` :: Rank -> Rank -> Bool

`allSuccRanks` *H* returns True iff all the cards in *H* have consecutive ranks.

`allSuccRanks` :: Hand -> Bool

31.2 Categorisation of hands

A `HandType` is one of (in order of increasing value):

1. `Garbage` – not any of the other kinds, worth only the ranks of its cards;
2. `Pair` – two cards have the same rank and all of the other cards have different ranks;
3. `TwoPair` – there are two pairs of different ranks and the other card is yet another rank.
4. `Triple` – three cards have the same rank and the rest other different ranks;
5. `Straight` – the cards all have sequential ranks (an ace can precede a deuce or follow a king) and some cards have different suits;
6. `Flush` – all cards are the same suit, but not with sequential ranks;
7. `FullHouse` – a triple and a pair;
8. `Poker` – four of a kind;
9. `StraightFlush` – all cards have sequential ranks and the same suit.

```
data HandType = Garbage | Pair | TwoPair | Triple  
              | Straight | Flush | FullHouse | Poker | StraightFlush  
              deriving (Eq, Ord, Show, Enum)
```

`handType` *H* classifies *H*.

`handType` :: Hand -> HandType

`isGarbage`, `isPair`, `isTwoPair`, `isStraight`, `isTriple`, `isStraight`, `isFlush`, `isFullHouse`, `isPoker` and `isStraightFlush` each return True iff some hand is that kind of hand.

```
isGarbage, isPair, isTwoPair, isTriple, isStraight,  
isFlush, isFullHouse, isPoker, isStraightFlush  
:: Hand -> Bool
```

31.3 Comparison of hands

The ordering of cards for the purpose of comparing hands is based solely on rank. Aces have the highest value.

instance Ord Card where

`compareCards` *cs cs'* compares two list of cards starting by comparing the highest ranked cards.

`compareCards` :: [Card] -> [Card] -> Ordering

`compareGroups` *css css'* compares lists of lists of cards, considering the corresponding lists in the order they come. Precondition: *css* and *css'* are the same length.

`compareGroups` :: [[Card]] -> [[Card]] -> Ordering

`compareHands` *H H'* compares two hands.

`compareHands` :: Hand -> Hand -> Ordering

H `beats` *H'* iff *H* is a better poker hand than *H'*. *H* `ties` *H'* if they have the same value.

`beats, ties` :: Hand -> Hand -> Bool

`better` *H H'* returns the better of hands *H* and *H'*.

`better` :: Hand -> Hand -> Hand

31.4 Hands with more than 5 cards

`best5` *H* picks the highest scoring 5 cards from *H*. Precondition *H* contains at least 5 cards.

`best5` :: Hand -> Hand

32 QuineMcClusky

Module `ABR.QuineMcClusky` implements the Quine-McClusky algorithm for simplifying boolean expressions as described in Rosen[5].

```
module ABR.QuineMcClusky (QMBit(..), qmSimplify) where
```

32.1 Data types

A Quine-McCluskey bit (`QMBit`) is either zero (`Zer`), one (`One`) or a placholding dash (`Dsh`). A list of them is a bit string. A list of bit strings is a formula. This module's purpose is the simplification of such a formula.

```
data QMBit = Zer | Dsh | One  
           deriving (Eq, Show)
```

32.2 Simplification

`qmSimplify` *bss* simplifies *bss*.

`qmSimplify` :: [[QMBit]] -> IO [[QMBit]]

32.3 Instance declarations

32.3.1 DeepSeq

```
instance DeepSeq QMBit where { }
```

33 Relational Database

Module `ABR.RelationalDB` is *UNDER CONSTRUCTION*.

```
module ABR.RelationalDB where
```

33.1 Definitions and data types

A *database* is a collection of tables. Each *table* represents a relation. Each row of a table represents one tuple, an element of the relation. Each column of a table, is an *attribute* of that table. The terms table and relation are synonymous, as are row and tuple.

Any given database, populated with data, is an instance of a *database schema*, a specification of the names and types of data in each table attribute.

A name is just a string. Names are used to identify particular attributes tables and databases.

```
type Name = String
```

An attribute specification consists of a name and a type.

```
data Attribute =  AString Name
                | AInteger Name
                | ADouble Name
                deriving (Eq, Ord, Show)
```

A *schema* consists of a name, which may be empty, and a list of specifications. A *table schema* contains a list of attribute specifications, and specifies the information to be stored in each column of a table. The relation schema that relation r is an instance of is denoted $\alpha(r)$. A database schema contains a list of table schemas.

```
data Schema a = Schema Name [a]
```

```
type TableSchema = Schema Attribute
```

```
type DatabaseSchema = Schema TableSchema
```

Each element of a row is one *datum*. The following union type allows each datum to be of one of a range of types. The types correspond to the possible types of attributes, with one extra to represent the absence of a datum.

```
data Datum =  DNull
              | DString String
              | DInteger Integer
              | DDouble Double
              deriving (Eq, Ord, Show)
```

Within each row, the order of the data is significant.

```
type Row = [Datum]
```

Each table has a schema that identifies the contents of each attribute.

```
data Table = Table TableSchema [Row]
```

33.2 Relational algebra

33.2.1 Projection of a tuple on a relation schema

The projection $t[X]$ of a tuple t on a relation schema X is computed by discarding the attributes in t that do not appear in X . This projection operation will be applied to many rows and is worth computing just once. A *projector* is a function that performs a projection on one tuple.

```
type Projector = Row -> Row
```

A projector can be computed from the new and old table schemas. `makeProjector` Y X returns either:

- **Nothing**, if the new schema Y contains an attribute not in the old schema X ; or
- **Just** (Y', p) , where table schema Y' is the same as the requested new table schema Y with the exception that the attributes are in the same order as in the old table schema X , and p is the projector that can be applied to tuple that conforms to X to produce a tuple the conforms to Y' .

```
makeProjector :: TableSchema -> TableSchema
              -> Maybe (TableSchema, Projector)
```

33.2.2 Projection of a relation on a relation schema

The projection $\pi_X(r)$ of a relation r on a relation schema X , is defined by:

1. $X \subseteq \alpha(r)$ and $\alpha(\pi_X(r)) = X$
2. $\pi_X(r) = \{t[X] : t \in r\}$

`projectTable` (X, p) r returns $\pi_X(r)$ using p to project all the tuples in r .

```
projectTable :: (TableSchema, Projector) -> Table -> Table
```

`proj` X r returns $\pi_X(r)$. If the schema restrictions are not met then the program will terminate with an error.

```
proj :: TableSchema -> Table -> Table
```

33.2.3 Natural join

The *natural join* $r_1 \bowtie r_2$ of relations r_1 and r_2 is defined by:

1. $\alpha(r_1 \bowtie r_2) = \alpha(r_1) \cup \alpha(r_2)$
2. $r_1 \bowtie r_2 = \{t \in (r_1 \times r_2) [\alpha(r_1 \bowtie r_2)] : t[\alpha(r_1)] \in r_1 \wedge t[\alpha(r_2)] \in r_2\}$

Each tuple of the joined relation contains the combined attributes from two tuples, one from each of the original relations. A joined tuple is only formed if the overlapping attributes were equal.

A *joiner* is a function that joins two tuples if the overlapping attributes are equal.

```
type Joiner = Row -> Row -> Maybe Row
```

`makeJoiner` X X' returns $(X \cup X', j)$, where j is a joiner that joins a tuple that conforms to X to a tuple that conforms to X' , producing a tuple conforming to $X \cup X'$ if the overlapping attributes are equal.

```
makeJoiner :: TableSchema -> TableSchema
           -> (TableSchema, Joiner)
```

`joinTables` (X, j) r_1 r_2 returns $r_1 \bowtie r_2$, provided $X = \alpha(r_1 \bowtie r_2)$. j is used to join individual tuples.

```
joinTables :: (TableSchema, Joiner) -> Table -> Table
           -> Table
```

$r_1 |><| r_1$ returns $r_1 \bowtie r_2$.

```
infixl 5 |><|
```

```
(|><|) :: Table -> Table -> Table
```

33.2.4 Union

The *union* $r_1 \cup r_2$ of relations r_1 and r_2 is defined by:

1. $\alpha(r_1) = \alpha(r_2)$ and $\alpha(r_1 \cup r_2) = \alpha(r_1)$
2. $r_1 \cup r_2 = \{t : t \in r_1 \vee t \in r_2\}$

r_1 ‘**u**’ r_2 returns $r_1 \cup r_2$, provided $\alpha(r_1) = \alpha(r_2)$. No check is performed to ensure this precondition.

```
u :: Table -> Table -> Table
```

33.2.5 Difference

The *difference* $r_1 - r_2$ of relations r_1 and r_2 is defined by:

1. $\alpha(r_1) = \alpha(r_2)$ and $\alpha(r_1 - r_2) = \alpha(r_1)$
2. $r_1 - r_2 = \{t : t \in r_1 \vee t \notin r_2\}$

r_1 ‘**dif**’ r_2 returns $r_1 - r_2$, provided $\alpha(r_1) = \alpha(r_2)$. No check is performed to ensure this precondition.

```
dif :: Table -> Table -> Table
```

33.2.6 Selection

The *selection* $\sigma_p(r)$ of relation r by predicate p is defined by:

1. $\alpha(\sigma_p(r)) = \alpha(r)$
2. $\sigma_p(r) = \{t \in r : p(r)\}$

`select` p r returns $\sigma_p(r)$.

```
select :: (Row -> Bool) -> Table -> Table
```

33.2.7 Renaming

The *renaming* $\rho_{B|A}(r)$ in relation r of attribute A to attribute B is defined by:

1. $A \in \alpha(r)$, $B \notin \alpha(r)$ and $\alpha(\rho_{B|A}(r)) = (\alpha(r) - \{A\}) \cup \{B\}$
2. $\rho_{B|A}(r) = \{t : t' \in r \wedge t'[B] = t[A] \wedge \forall C \in (\alpha(r) - \{A\}) \cdot t[C] = t'[C]\}$

`renameTable B A r` returns $\rho_{B|A}(r)$.

`renameTable :: Attribute -> Attribute -> Table -> Table`

`renameTableSchema B A X` returns $(X - \{A\}) \cup \{B\}$.

`renameTableSchema :: Attribute -> Attribute -> TableSchema -> TableSchema`

33.3 Datum operations

`addDatum :: Datum -> Datum -> Datum`

34 SendMail

Module `ABR.SendMail` lets a Haskell program send an email.

`module ABR.SendMail (sendMail) where`

34.1 Function

`sendMail` *whoTo* *subject content* sends an email to *whoTo* about *subject* containing *content*.

`sendMail :: String -> String -> String -> IO ()`

35 Showing

The `ABR.Showing` library provides functions to help write new instances of class `Show`, and to get control of numeric precision.

```
module ABR.Showing (
  showWithSep, showWithTerm, FormattedDouble(..),
  makeFormattedDouble, showFD
) where
```

35.1 Adding Delimiters

`showWithSep` *sep xs* shows the elements of *xs* separated by *sep*.

`showWithTerm` *term xs* shows the elements of *xs* terminated by terminator *term*. (Adapted from Mark Jones's Mini Prolog.)

```
showWithSep, showWithTerm ::
  Show a => String -> [a] -> ShowS
```

35.2 Controlling Precision

A `FormattedDouble` is a `Double` bound to a desired format. The format is one of: `FD` for no exponent; `ED` for an exponent; or `GD` for the best choice between the two. An optional integer specifies the number of digits after the decimal point.

```
data FormattedDouble =
  FD (Maybe Int) Double
  | ED (Maybe Int) Double
  | GD (Maybe Int) Double
```

This `Show` instance applies the formatting to the `Double`.

`instance Show FormattedDouble where`

```
showsPrec _ fd = case fd of
  FD md x -> showFFloat md x
  ED md x -> showEFloat md x
  GD md x -> showGFloat md x
```

`makeFormattedDouble` *format x* makes a `FormattedDouble` from a `Double` *x* and a string that describes the format, *format*, of the form `"f" | "e" | "g"` {digit}, e.g. `"f2"`.

```
makeFormattedDouble :: String -> Double -> FormattedDouble
makeFormattedDouble format x = (case format of
  "f" -> FD Nothing
  "e" -> ED Nothing
  "g" -> GD Nothing
  'f':cs -> FD (Just (read cs))
  'e':cs -> ED (Just (read cs))
  'g':cs -> GD (Just (read cs))
) x
```

`showFD` *format x* shows *x* using the format described by *format*.

```
showFD :: String -> Double -> String
showFD format x = show (makeFormattedDouble format x)
```

36 Supplies

Module `ABR.Supply` implements a name supply using a mutable variable in the IO monad.

```
module ABR.Supply (
  Supply, newSupply, supplyNext, peekNext
) where
```

36.1 Data types

A `Supply` is a value of any enumerated type.

`type Supply a =`

36.2 Creating a supply

`newSupply` *first* creates a new `Supply` that will commence with *first*.

`newSupply :: a -> IO (Supply a)`

36.3 Extracting values from a supply

`supplyNext` *supply* returns the next value from the *supply*.

`supplyNext :: Enum a => Supply a -> IO a`

`peekNext` *supply* returns the next value from the *supply*, but does not change the supply, so that the next value extracted will be the same.

`peekNext :: Supply a -> IO a`

37 Text.Configs

Module `ABR.Text.Configs` provides a type, parser and pretty printer for a sequence of configuration settings, as might be found in a configuration file. This kind of data could be stored in XML, but this format is nicer to edit by hand.

```
module ABR.Text.Configs (
  Config(..), Configs, configsL, configsP, stringL,
  showConfigs, read', lookupConfig, updateConfig,
  lookupParam, getParam, popTemplate
) where
```

37.1 Data types

A configuration, `Config`, is one of:

`CFlag` a flag that is set by its presence;

`CParam` a parameter with an associated value;

`CSet` a parameter with an associated set of configurations; or

`CList` a parameter with an associated list of configurations.

```
data Config =
  CFlag String
  | CParam String String
  | CSet String Configs
  | CList String [Configs]
  deriving (Eq, Ord)
```

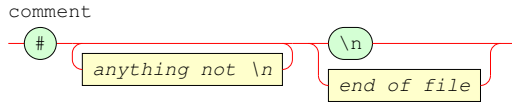
A `Configs` is a list of configurations.

`type Configs = [Config]`

37.2 Lexer

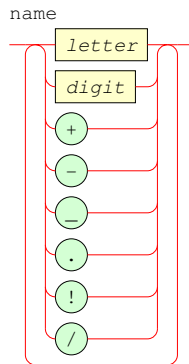
Comments in configuration files start with # and extend to the end of the line. Comments are treated as whitespace. There can be any amount of whitespace between tokens. Aside from inside strings, and to separate names, whitespace is not significant.

```
comment ::=
  "#" {$anything not \n$} ("\n" | $end of file$);
level="lexical".
```



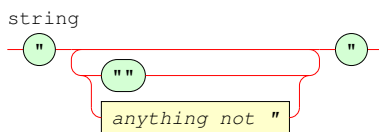
Names in configuration files may contain letters, digits, plus and minus signs, underscores, periods, bangs and slashes. Note that a number can lex as a name, as could many file paths. Names are case sensitive.

```
name ::= {$letter$ | $digit$ |
  "+" | "-" | "_" | "." | "!" | "/"};
level="lexical".
```



Strings are delimited by double quotes and may extend across many lines. Use two double quotes for one, à la Pascal.

```
string ::= "\"" {$anything not \"$} "\"";
level="lexical".
```



The other symbols used are:

= to bind a name to a value (either a name or a string), configuration set or configuration list;

{ to start a configuration set;

} to close a configuration set;

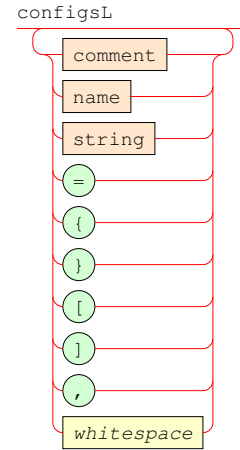
[to start a configuration list;

] to close a configuration list; and

, to separate items in a configuration list.

configsL is the lexer that will tokenize a configuration source.

```
configsL ::= {comment | name | string | "=" | "{" | "}"
  | "[" | "]" | "," | $whitespace$};
level="lexical".
```

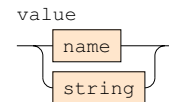


configsL :: Lexer

37.3 Parser

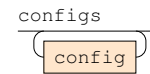
A value is either a name or a string.

```
value ::= name | string;
level="grammar".
```



A configs is a sequence of whitespace separated configs, parsed by **configsP**.

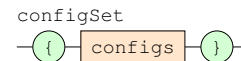
```
configs ::= {config};
level="grammar".
```



configsP :: Parser Configs

A configSet is a configs in braces.

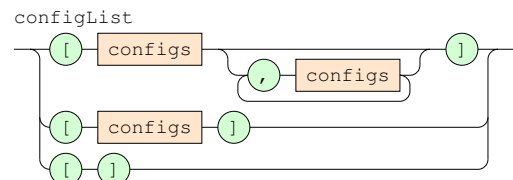
```
configSet ::= "{" configs "}";
level="grammar".
```



configSetP :: Parser Configs

A configList is a comma separated sequence of configs in brackets.

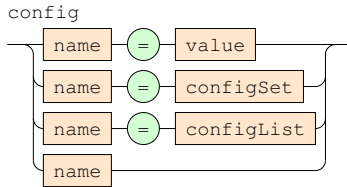
```
configList ::= "[" configs {"," configs} "]"
  | "[" configs "]"
  | "[" " ";
level="grammar".
```



configListP :: Parser [Configs]

A config is either: a binding of a name to a configSet, a configList, or a value; or just a name.

```
config ::= name "=" value
  | name "=" configSet
  | name "=" configList
  | name;
level="grammar".
```



- `_UNDEFINED_` if the `Config` does not exist;
- `_DEFINED_` if the `Config` is a flag;
- the unStringed text of a parameter;
- `_SET_` if the `Config` is a set; or
- `_LIST_` if the `Config` is a list.

37.4 Showing

instance Show Config where

`showConfigs` *cs* shows a list of configs.

`showConfigs :: Configs -> String`

37.5 Reading

`read'` *s* may be used to read a parameter value *s*, removing quotes first.

`read' :: Read a => String -> a`

37.6 Accessing

37.6.1 configPaths

A `configPath` is a string that selects a `Config` from within some `Configs`. It can be: the name of a `Config`, eg `name`; of the form `name.name` to select from inside a `CSet`; or of the form `name!digits.name` to select from inside a `CList`; or of combinations like `class!3.student!7.name.family`.

Note this description. While the names in a `Config` may be any double-quote-delimited string, those names are not useable in a `configPath`.

37.6.2 Lookup functions

`lookupConfig` *n cs* tries to find the named config *n* in *cs*, returning `Just` the first match or `Nothing`. *n* is a `configPath`.

`lookupConfig :: String -> Configs -> Maybe Config`

`updateConfig` *n c cs* tries to replace the named config *n* in *cs* with *c*. *n* is a `configPath`. If the named config does not exist it will be created. THIS IS INCOMPETELY IMPLEMENTED and is the wrong way to build `Configs` anyway.

`updateConfig :: String -> Config -> Configs -> Configs`

`lookupParam` *n cs* tries to find the named parameter *n* in *cs*, returning `Just` the first value or `Nothing`. (Note `elem` is all that is required to test for a flag.) *n* is a `configPath`.

`lookupParam :: String -> Configs -> Maybe String`

`getParam` *n cs* tries to find the named parameter. It is an error if the parameter can not be found. *n* is a `configPath`.

`getParam :: String -> Configs -> String`

37.7 Templates

`Configs` are a structured data type like XML and can be used to populate a template. The power of templates in part comes from being able to handle variations in the data that they might get populated with.

37.7.1 Simple Template Markup Language

- A template is a `String` containing any kind of text, marked up as HTML or anything else.
- The hash character (`#`) is the special escape character in a template. It is *always* treated specially.
- To output a hash, use two, eg `##`.
- The sequence `#configPath#` is replaced in the output by the value of the `Config`.
- The output value of a `Config` is:

- In the following sequences, no extra whitespace is permitted.
- The sequence `#ifdef#configPath#text#end#` outputs the *text* iff `configPath` leads to a `Config` of any kind that exists, otherwise outputs nothing. The *text* has any template markup in it processed as usual.
- The sequence `#ifundef#configPath#text#end#` outputs the *text* iff `configPath` does not lead a `Config` that exists, otherwise outputs nothing. The *text* has any template markup in it processed as usual.
- The sequence `#with#configPath#text#end#` outputs the *text* iff `configPath` leads to a set `Config` that exists. The *text* has any template markup in it processed using the set as the new root `Configs`. If the `configPath` does not lead to a `Config` that exists, the output is `_UNDEFINED_SET_`. If the `configPath` does not leads to a `Config` that exists, but is not a set, the output is `_NOT_A_SET_`.
- The sequence `#foreach#configPath#text#end#` outputs the *text* for each element of the list iff `configPath` leads to a list `Config` that exists. The *text* has any template markup in it processed using each element as its root `Configs`. If the `configPath` does not lead to a `Config` that exists, the output is `_UNDEFINED_LIST_`. If the `configPath` does not leads to a `Config` that exists, but is not a list, the output is `_NOT_A_LIST_`.

37.7.2 Populating templates

`popTemplate` *configs template* returns the text of the *template* populated with the data from the *configs*. NOT TESTED

`data Mode = Quiet | Print | Reprint`

`popTemplate :: Configs -> String -> String`

38 Text.Markup

Module `ABR.Text.Markup` is a collection of functions that operate on strings wrt to Markup Languages.

```
module ABR.Text.Markup (
    encodeHTML, encodeHTML', makeHTMLSafe, makeHTMLSafe',
    makeLatexSafe, latex2html
) where
```

38.1 Making text safe for HTML

`encodeHTML` *c* returns *c*'s special character encoding if *c* ∈ {<, >, &, "}, otherwise *c*. `encodeHTML'` *c* returns *c*'s special character encoding, additionally encoding all control characters.

`encodeHTML, encodeHTML' :: Char -> String`

`makeHTMLSafe` *cs* encodes all of the special characters in *cs*. It would be counter productive to put text containing tags through this filter. `makeHTMLSafe'` *cs* encodes all of the special characters in *cs* including all control characters.

`makeHTMLSafe, makeHTMLSafe' :: String -> String`

38.2 Making text safe for LaTeX

`makeLatexSafe` *cs* makes *cs* safe for inclusion in a `LATEX` document as plain text, by encoding some special characters.

`makeLatexSafe :: String -> String`

38.3 Converting LaTeX to HTML

`latex2html` *cs* converts L^AT_EX string *cs* to HTML. This is not meant for whole documents. It has some basics for writing comments just like this one. This is used by `mashdoc`.

```
latex2html :: String -> String
```

39 Text.String

Module `ABR.Text.String` is a collection of functions that operate on strings.

```
module ABR.Text.String (
  wordWrap, lJustify, rJustify, lJustify', rJustify',
  justifyColumn, makeTable, spaceColumns, makeTableL,
  makeTableMR, fields, unfields, nameLT, trim,
  fixNewlines, fixNewlines', spaces, findClosest,
  (++/++), (++/++), catenateWith, substs, subst,
  subHashNums, subHashNames, isCardinal, isFixed,
  isFloat, isSignedCardinal, isSignedFixed,
  isSignedFloat, unString, enString
) where
```

39.1 Word wrapping

`wordWrap` *width cs* wraps the words in *cs* to no wider than *width*, unless a word is wider than *width*, returning a list of lines.

```
wordWrap :: Int -> String -> [String]
```

39.2 Justification

`lJustify` *w cs* pads *cs* with extra spaces on the right to make the overall width not less than *w*. `rJustify` *w cs* pads *cs* with extra spaces on the left to make the overall width not less than *w*.

```
lJustify, rJustify :: Int -> String -> String
```

`lJustify'` *p w cs* pads *cs* with extra pad characters *p* on the right to make the overall width not less than *w*. `rJustify'` *p w cs* pads *cs* with extra pad characters *p* on the left to make the overall width not less than *w*.

```
lJustify', rJustify' :: Char -> Int -> String -> String
```

39.3 Tables with justified columns

`justifyColumn` *j col* justifies all of the strings in *col* using *j* to justify them all to the same width, which is the width of the widest string in *col*.

```
justifyColumn :: (Int -> String -> String) ->
  [String] -> [String]
```

`makeTable` *js cols* applies the justification functions in *js* to the corresponding columns in *cols* and assembles the final table. Short columns have extra blank rows added at the bottom.

```
makeTable :: [Int -> String -> String] ->
  [[String]] -> String
```

`spaceColumns` *cs cols* spaces out columns *cols* by inserting columns of replicated strings *cs*.

```
spaceColumns :: String -> [[String]] -> [[String]]
```

`makeTableL` *c cols* makes a table from *cols* using all left justification, with *c* used to pad columns and separate columns.

```
makeTableL :: Char -> [[String]] -> String
```

`makeTableMR` *js rows* makes a table from elements that are themselves multi-rowed. *js* is a list of justifiers for each column (as in `makeTable`). *rows* is a list of rows, where each row is a list of columns.

```
makeTableMR :: [Int -> String -> String] ->
  [[String]] -> String
```

39.4 Fields

These are functions for breaking a string into a list of fields and converting a list of fields into a string. The fields are delimited with a nominated special character. To permit the special character to appear in a field it is preceded by a nominated escape character. To permit the escape character to appear in a string, it is preceded by itself.

`fields` *d e cs* breaks string *cs* into a list of strings at each delimit character *d*, removing escape characters *e* where appropriate. If the escaping is not required use `ABR.List.chop` instead.

```
fields :: Char -> Char -> String -> [String]
```

`unfields` *d e css* converts *css* into one string, with each field separated by the delimit character *d*, adding escape characters *e* as needed. If the escaping is not required use `Data.List.intersperse` instead.

```
unfields :: Char -> Char -> [String] -> String
```

39.5 Whitespace

`trim` *cs* strips any whitespace from both ends of *cs*.

```
trim :: String -> String
```

`fixNewlines` *cs* rectifies the ends of lines in *cs*. It does not ensure that the last character is a newline.

```
fixNewlines :: String -> String
```

`fixNewlines'` *cs* rectifies the ends of lines in *cs*. This version ensures that the last line is complete, *i.e.* that unless *cs* is empty, the last character returned will be a newline.

```
fixNewlines' :: String -> String
```

`spaces'` *n* returns *n* spaces.

```
spaces :: Int -> String
```

39.6 Pattern matching and substitution

`findClosest` *pattern candidates* returns the position in *candidates* of the string which, ignoring case is closest to *pattern* or `-1` if *candidates* is empty.

```
findClosest :: String -> [String] -> Int
```

`subst` *prs cs* performs substitutions on *cs*. *prs* is a list of pairs (*p*, *r*), where *p* is a case sensitive pattern to be replaced by *r* wherever it occurs.

```
subst :: [(String, String)] -> String -> String
```

`subst` *p r cs* performs substitutions on *cs*. *p* is a case sensitive pattern to be replaced by *r* wherever it occurs.

```
subst :: String -> String -> String -> String
```

`subHashNums` *rs cs* performs substitutions on *cs*. *rs* is a list of replacements. *r*!#0 will replace the pattern `#0#`, *r*!#1 will replace the pattern `#1#`, ...

```
subHashNums :: [String] -> String -> String
```

`subHashNames` *nrs cs* performs substitutions on *cs*. *nrs* is a list of pairs (*n*, *r*), where *n* is a case sensitive name to be replaced by *r* wherever it occurs between a pair of hashes.

```
subHashNames :: [(String,String)] -> String -> String
```

39.7 Numbers

`isCardinal`, `isFixed`, `isFloat`, `isSignedCardinal`, `isSignedFixed` and `isSignedFloat` are all predicated that test whether a string could be parsed as a number.

```
isCardinal, isFixed, isFloat, isSignedCardinal,
  isSignedFixed, isSignedFloat :: String -> Bool
```

39.8 Names

`nameLT` $n_1 n_2$ returns `True` if name $n_1 < n_2$. Use this to sort names with `msort` when names are in *family-name comma other-names* format.

```
nameLT :: String -> String -> Bool
```

39.9 Path catenation operators

`++/++` joins two paths with a single `/`. `++.++` joins two paths with a single `.`. The utility of these operators is that any extra `/s` or `.s` at the join are removed.

```
infixl 6 ++/++, ++.++
```

```
(++/++), (++.++) :: String -> String -> String
```

More such operators can be constructed with

`catenateWith` $c cs cs'$, which catenates cs and cs' with exactly one c at the join.

```
catenateWith :: Char -> String -> String -> String
```

39.10 Simple String Delimitation

`unString` s rectifies string s , by removing the double quotes from each end (if present) and replacing pairs of double quotes with just one. If there are no double quotes in s , it is returned unchanged.

```
unString :: String -> String
```

`enString` s encodes a string with enclosing quotes and doubles any enclosed quotes.

```
enString :: String -> String
```

40 Versions

The `ABR.Versions` module provides replacements for `Prelude.readFile` and `Prelude.writeFile` that read the most recent and write the next version of a file. Each version of a file is distinguished by a *.number* extension appended to the root name of the file. It also provides some IO utilities.

```
module ABR.Versions (
  readLatest, writeNew, writeNew', writeNew'',
  purgeVersions, getNames, readFile', writeFile',
  removeR, createDirectory', removeVersions,
  latestDate
) where
```

40.1 Read the latest version

`readLatest` $dir root$ reads the contents of the latest version of the file with the given $root$ file name in directory dir . Either `Just` the contents are returned, or `Nothing` if no version of the file could be read.

```
readLatest :: FilePath -> FilePath -> IO (Maybe String)
```

40.2 Date of the latest version

`latestDate` $dir root$ returns a `String` containing the modification date of the latest version, if one exists.

```
latestDate :: FilePath -> FilePath -> IO (Maybe String)
```

40.3 Write the next version

`writeNew` $dir root content$ writes $content$ to a new version of the file with the given $root$ file name in directory dir . The new file is assigned the access permissions of dir (masked by `-rw-rw-rw-`) and the same `userID` and `groupID` as dir .

```
writeNew :: FilePath -> FilePath -> String -> IO ()
```

`writeNew'` $dir root content mode$ writes $content$ to a new version of the file with the given $root$ file name in directory dir . The new file has the given access $mode$ and the same `userID` and `groupID` as dir .

```
writeNew' ::
  FilePath -> FilePath -> String -> FileMode -> IO ()
```

`writeNew''` $dir root content$ writes $content$ to a new version of the file with the given $root$ file name in directory dir . The new file has the access mode `-rw-----` and the same `userID` and `groupID` as dir .

```
writeNew'' :: FilePath -> FilePath -> String -> IO ()
```

40.4 Purge old versions

`purgeVersions` $dir root$ deletes all old versions of the file with the given $root$ name in directory dir .

```
purgeVersions :: FilePath -> FilePath -> IO ()
```

40.5 Remove all versions

`removeVersions` $dir root$ deletes all versions of the file with the given $root$ name in directory dir .

```
removeVersions :: FilePath -> FilePath -> IO ()
```

40.6 Get all versions

`getNames` $dir root$ returns the list of filenames in directory dir that contain the given $root$ file name and a version number.

```
getNames :: FilePath -> FilePath -> IO [String]
```

40.7 Read and write file bottlenecks

`readFile'` $path$ provides a safe way to read a file without raising an exception if the file does not exist. It returns `Nothing` if the file could not be read, `Just contents` otherwise.

```
readFile' :: FilePath -> IO (Maybe String)
```

`writeFile'` $dir file content$ writes $content$ to $dir/file$. The new file is assigned the access permissions of dir (masked by `-rw-rw-rw-`) and the same `userID` and `groupID` as dir .

```
writeFile' :: FilePath -> FilePath -> String -> IO ()
```

40.8 Creating and removing directories

`removeR` $path$ removes a file or directory, $path$, first, recursively removing all its contents if it is a directory. If $path$ does not exist, nothing is done.

```
removeR :: FilePath -> IO ()
```

`createDirectory'` $parentDir newDir$ creates a directory, $newDir$ inside $parentDir$. If $newDir$ already exists (as a file or directory) it is removed (along with its contents) first. $newDir$ is assigned the same `userID` `groupID` and access permissions as $parentDir$.

```
createDirectory' :: FilePath -> FilePath -> IO ()
```

References

- [1] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999. 21
- [2] J. Launchbury. Graph algorithms with a functional flavour. In Jeuring and Meijer [6], pages 308–331. 21
- [3] Graham Hutton. Higher-order functions for parsing. *J. Functional Programming*, 2:323–343, 1992. 27
- [4] Jeroen Fokker. Functional parsers. In Jeuring and Meijer [6], pages 1–23. 27
- [5] K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 5th edition, 2005. 32
- [6] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995. 2, 4

Index

*>, 19
*?, 5
*%>, 19
++-++, 12
++.++, 27
++/++, 27
+::, 7
+?, 5
.*, 8
.*, 8
.+, 8
.-, 8
.<, 8
.<-, 8
.<=, 8
. &, 8
. |, 8
<*, 19
<*>, 19
<*>, 19
<+>, 19
<|>, 19
??, 5
@>, 19
#>, 19
#?, 5
\$!!, 11
%>, 19
&?, 5
!!!, 10

ABR.Args, 3
ABR.CGI, 4
ABR.Control.Check, 4
ABR.Control.List, 5
ABR.Data.BSTree, 5, 11
ABR.Data.HashTables, 6
ABR.Data.List, 6
ABR.Data.NameTable, 7
ABR.Data.Queue, 8
ABR.Data.Set, 8
ABR.Data.SparseSet, 9
ABR.Daytime, 9
ABR.Debug.Array, 10
ABR.Debug.IArray, 10
ABR.DeepSeq, 11
ABR.EPS, 11
ABR.Geometry, 13
ABR.Graphs, 13
ABR.HaskellLexer, 14
ABR.LockFile, 15
ABR.MySQL, 17
ABR.MySQLCBinding, 15
ABR.Parser, 18
ABR.Parser.Checks, 21
ABR.Parser.Lexers, 20
ABR.Parser.Pos, 18
ABR.PlayingCards, 21
ABR.Poker, 21
ABR.QuineMcClusky, 22
ABR.RelationalDB, 22
ABR.SendMail, 23
ABR.Showing, 24
ABR.Supply, 24
ABR.Text.Configs, 24
ABR.Text.Markup, 26
ABR.Text.String, 26
ABR.Versions, 28
ABRHLibs.tar.gz, 3
accumArray', 10
Ace, 21
addressE, 4
addressN, 4
aE, 4
allSuccRanks, 22
allUnique, 7

alsoNotSat, 19
alsoSat, 19
Analyser, 18
Angle, 13
arc, 12
areaE, 4
areAnyLocked, 15
areSuccRanks, 22
array', 10
attachQ, 8

bagElem, 7
baseE_, 4
bE, 4
beats, 22
best5, 22
better, 22
bigE, 4
bigN, 4
blockGuard, 15
blockquoteE, 4
blockquoteN, 4
bN, 4
bodyE, 4
bodyN, 4
Box, 13
box, 12
BPS, 11
bpsToEps, 11
brE_, 4
brN_, 4
BSTree, 5

captionE, 4
captionN, 4
Card, 21
card, 8
cardinalL, 20
cartProd, 6
catenateWith, 27
centerE, 4
centerN, 4
CFlag, 24
Check, 5
CheckFail, 5
checkLex, 21
checkParse, 21
CheckPass, 5
CheckResult, 5
chop, 7
chops, 7
citeE, 4
citeN, 4
CList, 24
closepath, 12
Clubs, 21
codeE, 4
codeN, 4
Col, 18
combinations, 6
compareCards, 22
compareGroups, 22
compareHands, 22
compiling, 3
Config, 24
Configs, 24
configsL, 25
configsP, 25
cons, 19
Could, 18
countBST, 5
countSS, 9
CParam, 24
createDirectory', 28
CSet, 24

dataSatisfies, 19
dataSatisfies', 19

dayAndTimeP, 10
 Daytime, 9
 daytimeL, 9
 daytimeP, 10
 ddE, 4
 ddN, 4
 Deck, 21
 deck52, 21
 deck54, 21
 declarations, 14
 declared, 14
 DeepSeq, 11
 deepSeq, 11
 deleteBST, 5
 deleteSS, 9
 deliterate, 14
 depthBST, 5
 detachQ, 8
 Diamonds, 21
 diffSS, 9
 dirE, 4
 dirN, 4
 discardInners, 14
 disjoint, 7
 distAroundOval, 13
 distribution, 3
 divE, 4
 dlE, 4
 dlN, 4
 docType, 4
 domBST, 5
 dropEach, 6
 dropWhite, 20
 Dsh, 22
 dtE, 4
 dtHrs, 9
 dtMins, 9
 dtN, 4
 dtSecs, 9
 dumpFormData, 4
 dumpHT, 6
 duplicates, 7

 ED, 24
 elemSS, 9
 emE, 4
 emN, 4
 emptyBST, 5
 emptyQ, 8
 emptySS, 9
 encodeHTML, 26
 encodeHTML', 26
 endA, 18
 endL, 20
 enString, 27
 Enum_mysql_option, 15
 EPS, 11
 epsDraw, 11
 epsilonA, 18
 errMsg, 20
 Error, 18
 errorA, 18
 eset, 8
 extractQ, 8

 Fail, 18
 failA, 18
 FD, 24
 fields, 27
 findClosest, 27
 findOpts, 3
 findSubset, 7
 fixedL, 21
 fixNewlines, 27
 fixNewlines', 27
 FlagMinus, 3
 FlagPlus, 3
 FlagS, 3

 flattenBST, 5
 flattenSS, 9
 floatL, 21
 Flush, 22
 FontBlock, 12
 fontE, 4
 FontString, 12
 FontTag, 12
 FormattedDouble, 24
 formE, 4
 formfeedL, 20
 fragments, 6
 fragments', 6
 Friday, 9
 frontQ, 8
 fsWidth, 12
 ftWidth, 12
 FullHouse, 22

 Garbage, 22
 GD, 24
 GeoNum, 13
 getCONTENT_LENGTH, 4
 getFormData, 4
 getFormData', 4
 getNames, 28
 getParam, 26
 getPATH_INFO, 4
 getPos, 18
 getQUERY_STRING, 4
 getSCRIPT_NAME, 4
 getScriptDirectory, 4
 grestore, 12
 groupByRank, 22
 groupBySuit, 22
 gsave, 12

 h1E, 4
 h1N, 4
 h2E, 4
 h2N, 4
 h3E, 4
 h3N, 4
 h4E, 4
 h4N, 4
 h5E, 4
 h5N, 4
 h6E, 4
 h6N, 4
 Hand, 21
 HandType, 22
 handType, 22
 HashTable, 6
 HasPos, 18
 HAttributes, 4
 headE, 4
 headN, 4
 Hearts, 21
 Helvetica10, 12
 Helvetica10Bold, 12
 Helvetica10BoldOblique, 12
 helvetica10BoldObliqueWidth, 12
 helvetica10BoldWidth, 12
 Helvetica10Oblique, 12
 helvetica10ObliqueWidth, 12
 helvetica10Width, 12
 hrE-, 4
 hrN-, 4
 HTag, 4
 htmlE, 4
 htmlN, 4
 htmlT, 4

 iBox, 13
 iE, 4
 iGeo, 13
 imgE-, 4
 iN, 4

- inInterval, 10
- inputE_, 4
- insertName, 7
- insertSS, 9
- insetBox, 13
- insetSeg, 13
- interleave, 6
- iPoint, 13
- isCardinal, 27
- isEmptyQ, 8
- isFixed, 27
- isFloat, 27
- isFlush, 22
- isFullHouse, 22
- isGarbage, 22
- isindexE_, 4
- isindexN_, 4
- isLockedFile, 15
- isPair, 22
- isPoker, 22
- isProperSubset, 7
- isSignedCardinal, 27
- isSignedFixed, 27
- isSignedFloat, 27
- isStraight, 22
- isStraightFlush, 22
- isSubSequence, 7
- isSubSet, 9
- isSubset, 7
- isTriple, 22
- isTwoPair, 22

- Jack, 21
- joinBPS, 11
- Joker, 21
- justifyColumn, 27

- kbdE, 4
- kbdN, 4
- King, 21

- latestDate, 28
- latex2html, 26
- leastRightShift, 13
- leftBST, 5
- Lexeme, 19
- Lexer, 19
- liE, 4
- liN, 4
- Line, 13, 18
- lineNo, 20
- LineSeg, 13
- lineto, 12
- linkE_, 4
- list, 8
- list2BST, 5
- list2SS, 9
- listL, 20
- literalL, 19
- literalP, 20
- lJustify, 27
- lJustify', 27
- lockFile, 15
- lockFiles, 15
- lockGuard, 15
- lookupBST, 5
- lookupConfig, 26
- lookupFlag, 3
- lookupGuard, 5
- lookupHT, 6
- lookupName, 7
- lookupParam, 3, 26
- lookupQueue, 3

- Makefile, 3
- MakeFontTags, 12
- makeFontTags, 12
- makeFontTagsPrec, 12

- makeFormattedDouble, 24
- makeHTMLSafe, 26
- makeLatexSafe, 26
- makeNameArray, 8
- makeTable, 27
- makeTableL, 27
- makeTableMR, 27
- many, 19
- manyUntil, 19
- mapBST, 5
- mapE, 4
- meet, 7
- memberBST, 5
- menuE, 4
- menuN, 4
- merge, 6
- metaE_, 4
- mimeHeader, 4
- mkSS, 9
- mnub, 7
- moduleName, 14
- Monday, 9
- moveto, 12
- Msg, 18
- msort, 6
- My_bool, 15
- My_ulonglong, 15
- myClose, 18
- myConnect, 17
- myFetch, 18
- myQuery, 18
- MYSQL, 15
- MySQL, 17
- mysql_affected_rows, 15
- mysql_change_user, 15
- mysql_character_set_name, 16
- mysql_close, 16
- mysql_data_seek, 16
- mysql_errno, 16
- mysql_error, 16
- mysql_fetch_field, 16
- mysql_fetch_field_direct, 16
- mysql_fetch_fields, 16
- mysql_fetch_lengths, 16
- mysql_fetch_row, 16
- MYSQL_FIELD, 15
- mysql_field_count, 16
- MYSQL_FIELD_OFFSET, 15
- mysql_field_seek, 16
- mysql_field_tell, 16
- mysql_free_result, 16
- mysql_get_client_info, 16
- mysql_get_host_info, 16
- mysql_get_proto_info, 16
- mysql_get_server_info, 16
- mysql_info, 16
- mysql_init, 16
- mysql_insert_id, 16
- mysql_kill, 16
- mysql_list_dbs, 16
- mysql_list_fields, 16
- mysql_list_processes, 16
- mysql_list_tables, 16
- mysql_num_fields, 16
- mysql_num_rows, 16
- mysql_options, 17
- mysql_ping, 17
- mysql_query, 17
- mysql_real_connect, 17
- mysql_real_escape_string, 17
- mysql_real_query, 17
- MYSQL_RES, 15
- MYSQL_ROW, 15
- MYSQL_ROW_OFFSET, 15
- mysql_row_seek, 17
- mysql_row_tell, 17
- mysql_select_db, 17
- mysql_shutdown, 17

mysql_stat, 17
mysql_store_result, 17
mysql_thread_id, 17
mysql_use_result, 17

NameArray, 7
nameLT, 27
NameTable, 7
netBox, 13
newHT, 6
newlineL, 20
newNameTable, 7
newpath, 12
newSupply, 24
nextidE., 4
nofail, 19
nofail', 19
noSuperSets, 7
notElemSS, 9
notSubSequence, 7
nullBST, 5
nullSS, 9

odiff, 7
offside, 14
OK, 18
olE, 4
olN, 4
One, 22
optional, 19
optionE, 4
Options, 3
OptSpec, 3
OptVal, 3
osect, 7
ounion, 7

Pair, 22
pairs2BST, 5
pam, 5
ParamMissingValue, 3
ParamQueue, 3
ParamS, 3
ParamValue, 3
Parser, 20
pE, 4
peekNext, 24
permutations, 6
permutations', 6
pN, 4
Point, 13
Poker, 22
popTemplate, 26
Pos, 18
pos, 18
powSet, 6
powSet', 6
powSet_ge1, 6
powSet_ge1', 6
pPlus, 7
precedes, 18
preE, 4
preLex, 19
preN, 4
printDocType, 4
printMimeHeader, 4
programL, 14
promoteMethods, 14
properSublists, 6
PS, 11
psStr, 12
purgeVersions, 28
put, 4
put', 4

QMBit, 22
qmSimplify, 22
Queen, 21

Queue, 8
QueueS, 3

R10, 21
R2, 21
R3, 21
R4, 21
R5, 21
R6, 21
R7, 21
R8, 21
R9, 21
Rank, 21
rank, 21
ranks, 21
read', 25
readFile', 28
readLatest, 28
removeR, 28
removeVersions, 28
returnL, 20
rightBST, 5
rJustify, 27
rJustify', 27

sall, 8
sampE, 4
sampN, 4
sany, 8
satisfyL, 19
Saturday, 9
sectSS, 9
segToLine, 13
select, 8
selectE, 4
sendMail, 24
separate, 6
set, 8
set1, 8
setUpFonts, 11
sfoldl, 8
sfoldl1, 8
sfoldr, 8
sfoldr1, 8
shiftBoxes, 13
show., 12
showConfigs, 25
showDT24, 10
showFD, 24
showWithSep, 24
showWithTerm, 24
shuffle, 21
signedCardinalL, 21
signedFixedL, 21
SimpleLit, 3
smalle, 4
smallN, 4
smap, 8
snub, 7
snull, 8
soft, 20
some, 19
someUntil, 19
sortByLength, 6
sortByRankSuit, 22
sortBySuitRank, 22
Space, 12
spaceColumns, 27
spaceL, 20
spaces', 27
Spades, 21
SparseSet, 9
split, 6
sprod, 8
ssect, 8
Straight, 22
StraightFlush, 22
stringL, 20

stroke, 12
strongE, 4
strongN, 4
styleE, 4
subBag, 7
subE, 4
subHashNames, 27
subHashNums, 27
subN, 4
subsSuffix, 7
subst, 27
substs, 27
succeedA, 18
Suit, 21
suit, 21
suits, 21
Sunday, 9
sunion, 8
supE, 4
supN, 4
Supply, 24
supplyNext, 24
Symbol10, 12
symbol10Width, 12

tabL, 20
tableE, 4
tableN, 4
Tag, 19
tagFilter, 20
tagP, 20
tdE, 4
tdN, 4
textareaE, 4
thE, 4
thN, 4
Thursday, 9
ties, 22
Times10, 12
Times10Ital, 12
times10ItalWidth, 12
times10Width, 12
titleE, 4
titleN, 4
TLP, 19
TLPs, 19
tokenL, 20
tomorrow, 10
total, 19
translate, 12
trE, 4
trim, 27
trim2, 7
trimN, 7
Triple, 22
trN, 4
ttE, 4
ttN, 4
Tuesday, 9
TwoPair, 22

uE, 4
ulE, 4
ulN, 4
uN, 4
unfields, 27
unionSS, 9
unlex, 14
unlockFile, 15
unlockFiles, 15
unset1, 8
unString, 27
updateBST, 5
updateConfig, 26
updateHT, 6

varE, 4
varN, 4

vertabl, 20

warnMsg, 20
Wednesday, 9
Weekday, 9
weekdayP, 9
whitespaceL, 20
wordWrap, 26
wrapToAspect, 12
wrapWithinWidth, 12
writeFile', 28
writeNew, 28
writeNew', 28
writeNew'', 28

yesterday, 10

Zer, 22