Abstract

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

Contents

1 Introduction 4

2 Installation 4
  2.1 Dependencies 4
  2.2 Building 4

3 Util.Args 4
  3.1 Data types 4
  3.2 Empty options 4
  3.3 Option detection 4
  3.4 Adding and deleting options 4
  3.5 Looking up options 5
  3.6 Filename globbing 5

4 Util.Errors 5
  4.1 Warning and error data type 5
  4.2 Warning and error category type class 5
  4.3 Data types 5
  4.4 Returning successfully, failing or warning 5
  4.5 Instances 5
    4.5.1 Functor 5
    4.5.2 Applicative 5
    4.5.3 Monad 5

5 Util.Open 5
  5.1 Opening a URL 5

6 Util.Pos 5
  6.1 Positions in a source 5
  6.2 Overloaded projector 6
    6.2.1 Container instances 6
  6.3 Relative positions 6

7 Util.Time 6
  7.1 Data types 6
  7.2 Time conversions 6
  7.3 Getting the current time 6
  7.4 Format strings 6
  7.5 Formatting times 6
  7.6 Formated current time 6
  7.7 Formated file modification time 6
  7.8 Overloading for old/new time systems 6
  7.9 Instances 6
    7.9.1 LegacyTimes 6

8 Control.Check 7
  8.1 Data type 7
  8.2 Sequencing checks 7
  8.3 Parallel checks 7

9 Control.Map 7
  9.1 lookupGuard 7

10 Data.List 7
  10.1 Sorting 7
  10.2 Combinatorics 7
  10.3 Bag-like operations 8
  10.4 Set-like operations 8
  10.5 Subsequence operations 8

11 Data.NameTable 9
  11.1 Data types 9
  11.2 Creating a name table 9
  11.3 Looking up by name 9
  11.4 Creating a name array 9

12 Data.Supply 9
  12.1 Data types 9
  12.2 Creating a supply 9
  12.3 Extracting values from a supply 9

13 Debug.Array 9
  13.1 Functions 9

14 Debug.IArray 9
  14.1 Functions 9

15 Parser 9
  15.1 Error messages 9
  15.2 Results 10
  15.3 Analysers 10
  15.4 Elementary analysers 10
  15.5 Elementary analyser combinators 10
  15.6 Analyser result modifiers 10
  15.7 More analyser combinators 10
  15.8 Lexers 10
  15.9 Elementary lexers 11
  15.10 Special combinators for lexers 11
  15.11 Frequently used lexers 11
  15.12 Parsing 11
  15.13 Elementary parsers 11
  15.14 Parser result modifiers 11
  15.15 Error reporting 11
  15.16 Instance declarations 11

16 Parser.Lexers 11
  16.1 Frequently used lexers 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 Parser.Checks</td>
<td>12</td>
</tr>
<tr>
<td>17.1 Easy lexer and parser sequencing</td>
<td>12</td>
</tr>
<tr>
<td>18 Parser.Predicates</td>
<td>12</td>
</tr>
<tr>
<td>18.1 IsNumber predicates</td>
<td>12</td>
</tr>
<tr>
<td>19 Text.Configs</td>
<td>13</td>
</tr>
<tr>
<td>19.1 Data types</td>
<td>13</td>
</tr>
<tr>
<td>19.2 Lexer</td>
<td>14</td>
</tr>
<tr>
<td>19.3 Parser</td>
<td>14</td>
</tr>
<tr>
<td>19.4 Showing</td>
<td>15</td>
</tr>
<tr>
<td>19.5 Reading</td>
<td>15</td>
</tr>
<tr>
<td>19.6 Accessing</td>
<td>15</td>
</tr>
<tr>
<td>19.6.1 configPaths</td>
<td>15</td>
</tr>
<tr>
<td>19.6.2 Lookup functions</td>
<td>15</td>
</tr>
<tr>
<td>19.7 Templates</td>
<td>15</td>
</tr>
<tr>
<td>19.7.1 Simple Template Markup Language</td>
<td>15</td>
</tr>
<tr>
<td>19.7.2 Populating templates</td>
<td>15</td>
</tr>
<tr>
<td>20 Text.JSON</td>
<td>15</td>
</tr>
<tr>
<td>20.1 Basic data types</td>
<td>15</td>
</tr>
<tr>
<td>20.2 Construction</td>
<td>16</td>
</tr>
<tr>
<td>20.3 Lexer</td>
<td>16</td>
</tr>
<tr>
<td>20.4 Parser</td>
<td>16</td>
</tr>
<tr>
<td>20.5 Interrogation</td>
<td>16</td>
</tr>
<tr>
<td>20.6 Instances</td>
<td>16</td>
</tr>
<tr>
<td>20.6.1 Showing</td>
<td>16</td>
</tr>
<tr>
<td>20.6.2 Interrogation</td>
<td>16</td>
</tr>
<tr>
<td>21 Text.CSV</td>
<td>16</td>
</tr>
<tr>
<td>21.1 Basic data types</td>
<td>16</td>
</tr>
<tr>
<td>21.2 Parser</td>
<td>16</td>
</tr>
<tr>
<td>21.3 Interrogation</td>
<td>16</td>
</tr>
<tr>
<td>21.4 Instances</td>
<td>16</td>
</tr>
<tr>
<td>21.4.1 Showing</td>
<td>16</td>
</tr>
<tr>
<td>21.4.2 Interrogation</td>
<td>16</td>
</tr>
<tr>
<td>22 Text.Markup</td>
<td>16</td>
</tr>
<tr>
<td>22.1 Making text safe for HTML</td>
<td>16</td>
</tr>
<tr>
<td>22.2 Making text safe for LaTeX</td>
<td>17</td>
</tr>
<tr>
<td>22.3 Converting LaTeX to HTML</td>
<td>17</td>
</tr>
<tr>
<td>23 Text.String</td>
<td>17</td>
</tr>
<tr>
<td>23.1 Word wrapping</td>
<td>17</td>
</tr>
<tr>
<td>23.2 Justification</td>
<td>17</td>
</tr>
<tr>
<td>23.3 Tables with justified columns</td>
<td>17</td>
</tr>
<tr>
<td>23.4 Fields</td>
<td>17</td>
</tr>
<tr>
<td>23.5 Whitespace</td>
<td>17</td>
</tr>
<tr>
<td>23.6 Pattern matching and substitution</td>
<td>17</td>
</tr>
<tr>
<td>23.7 Names</td>
<td>17</td>
</tr>
<tr>
<td>23.8 Path catenation operators</td>
<td>18</td>
</tr>
<tr>
<td>23.9 Simple String Delimitation</td>
<td>18</td>
</tr>
<tr>
<td>24 Text.Showing</td>
<td>18</td>
</tr>
<tr>
<td>24.1 Adding Delimiters</td>
<td>18</td>
</tr>
<tr>
<td>24.2 Controlling Precision</td>
<td>18</td>
</tr>
<tr>
<td>25 Logic.Kinds</td>
<td>18</td>
</tr>
<tr>
<td>25.1 Data type</td>
<td>18</td>
</tr>
<tr>
<td>25.2 Unification</td>
<td>18</td>
</tr>
<tr>
<td>25.3 Kind inference</td>
<td>18</td>
</tr>
<tr>
<td>25.4 Instance declarations</td>
<td>18</td>
</tr>
<tr>
<td>25.4.1 Showing</td>
<td>18</td>
</tr>
<tr>
<td>26 Logic.Constans</td>
<td>18</td>
</tr>
<tr>
<td>26.1 Data types</td>
<td>18</td>
</tr>
<tr>
<td>26.2 Parsers</td>
<td>19</td>
</tr>
<tr>
<td>26.3 Collecting constants</td>
<td>19</td>
</tr>
<tr>
<td>26.4 Instance declarations</td>
<td>19</td>
</tr>
<tr>
<td>26.4.1 Comparing</td>
<td>19</td>
</tr>
<tr>
<td>26.4.2 Positions</td>
<td>19</td>
</tr>
<tr>
<td>26.4.3 Showing</td>
<td>19</td>
</tr>
<tr>
<td>26.4.4 Collecting constants</td>
<td>19</td>
</tr>
<tr>
<td>26.4.5 Kind inference</td>
<td>19</td>
</tr>
<tr>
<td>26.4.6 DeepSeq</td>
<td>19</td>
</tr>
<tr>
<td>27 Logic.Variables</td>
<td>19</td>
</tr>
<tr>
<td>27.1 Data type</td>
<td>19</td>
</tr>
<tr>
<td>27.2 Parsers</td>
<td>19</td>
</tr>
<tr>
<td>27.3 Collecting variables</td>
<td>19</td>
</tr>
<tr>
<td>27.4 Grounding</td>
<td>19</td>
</tr>
<tr>
<td>27.5 Instance declarations</td>
<td>19</td>
</tr>
<tr>
<td>27.5.1 Comparing</td>
<td>19</td>
</tr>
<tr>
<td>27.5.2 Positions</td>
<td>19</td>
</tr>
<tr>
<td>27.5.3 Showing</td>
<td>19</td>
</tr>
<tr>
<td>27.5.4 Collecting variables</td>
<td>20</td>
</tr>
<tr>
<td>27.5.5 Kind inference</td>
<td>20</td>
</tr>
<tr>
<td>27.5.6 Kind inference</td>
<td>20</td>
</tr>
<tr>
<td>28 Logic.Arguments</td>
<td>20</td>
</tr>
<tr>
<td>28.1 Data type</td>
<td>20</td>
</tr>
<tr>
<td>28.2 Parsers</td>
<td>20</td>
</tr>
<tr>
<td>28.3 Instance declarations</td>
<td>20</td>
</tr>
<tr>
<td>29 Logic.Atoms</td>
<td>20</td>
</tr>
<tr>
<td>29.1 Data type</td>
<td>20</td>
</tr>
<tr>
<td>29.2 Parsers</td>
<td>20</td>
</tr>
<tr>
<td>29.3 Collecting atoms</td>
<td>21</td>
</tr>
<tr>
<td>29.4 Instance declarations</td>
<td>21</td>
</tr>
<tr>
<td>29.4.1 Comparing</td>
<td>21</td>
</tr>
<tr>
<td>29.4.2 Positions</td>
<td>21</td>
</tr>
<tr>
<td>29.4.3 Showing</td>
<td>21</td>
</tr>
<tr>
<td>29.4.4 Collecting Atoms</td>
<td>21</td>
</tr>
<tr>
<td>29.4.5 Collecting constants</td>
<td>21</td>
</tr>
<tr>
<td>29.4.6 Collecting variables</td>
<td>21</td>
</tr>
<tr>
<td>29.4.7 Grounding</td>
<td>21</td>
</tr>
<tr>
<td>29.4.8 DeepSeq</td>
<td>21</td>
</tr>
<tr>
<td>30 Logic.Literals</td>
<td>21</td>
</tr>
<tr>
<td>30.1 Data type</td>
<td>21</td>
</tr>
<tr>
<td>30.2 Parser</td>
<td>21</td>
</tr>
<tr>
<td>30.3 Negation</td>
<td>21</td>
</tr>
<tr>
<td>30.4 Instance declarations</td>
<td>21</td>
</tr>
<tr>
<td>30.4.1 Comparing</td>
<td>21</td>
</tr>
<tr>
<td>30.4.2 Positions</td>
<td>21</td>
</tr>
<tr>
<td>30.4.3 Showing</td>
<td>21</td>
</tr>
<tr>
<td>30.4.4 Collecting Atoms</td>
<td>21</td>
</tr>
<tr>
<td>30.4.5 Collecting constants</td>
<td>21</td>
</tr>
<tr>
<td>30.4.6 Collecting variables</td>
<td>21</td>
</tr>
<tr>
<td>30.4.7 Collecting variables</td>
<td>22</td>
</tr>
<tr>
<td>30.4.8 Grounding</td>
<td>22</td>
</tr>
<tr>
<td>30.4.9 DeepSeq</td>
<td>22</td>
</tr>
<tr>
<td>31 Logic.QuineMcClusky</td>
<td>22</td>
</tr>
<tr>
<td>31.1 Data types</td>
<td>22</td>
</tr>
<tr>
<td>31.2 Simplification</td>
<td>22</td>
</tr>
<tr>
<td>31.3 Instance declarations</td>
<td>22</td>
</tr>
<tr>
<td>31.3.1 DeepSeq</td>
<td>22</td>
</tr>
<tr>
<td>32 Graphics.Geometry</td>
<td>22</td>
</tr>
<tr>
<td>32.1 Data types</td>
<td>22</td>
</tr>
<tr>
<td>32.2 Geometric computations</td>
<td>22</td>
</tr>
<tr>
<td>32.3 Instance declarations</td>
<td>23</td>
</tr>
<tr>
<td>33 Graphics.EPS</td>
<td>23</td>
</tr>
<tr>
<td>33.1 Data types</td>
<td>23</td>
</tr>
<tr>
<td>33.2 Finalizing to EPS</td>
<td>23</td>
</tr>
<tr>
<td>33.3 Drawing in EPS</td>
<td>23</td>
</tr>
<tr>
<td>33.4 Merging EPS components</td>
<td>23</td>
</tr>
<tr>
<td>33.5 Drawing text</td>
<td>23</td>
</tr>
<tr>
<td>33.5.1 Switching fonts efficiently</td>
<td>23</td>
</tr>
<tr>
<td>33.5.2 Font metrics</td>
<td>23</td>
</tr>
<tr>
<td>33.5.3 Font tags</td>
<td>24</td>
</tr>
<tr>
<td>33.5.4 Font strings</td>
<td>24</td>
</tr>
<tr>
<td>33.5.5 Font blocks</td>
<td>24</td>
</tr>
<tr>
<td>33.5.6 Text encoding</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>33.6</td>
<td>Conveniences</td>
</tr>
<tr>
<td>33.7</td>
<td>Instance declarations</td>
</tr>
<tr>
<td>33.7.1</td>
<td>EPSDrawable</td>
</tr>
<tr>
<td>34</td>
<td>File.Lock</td>
</tr>
<tr>
<td>34.1</td>
<td>Basic lock operations</td>
</tr>
<tr>
<td>34.2</td>
<td>Multiple file operations</td>
</tr>
<tr>
<td>34.3</td>
<td>Guards</td>
</tr>
<tr>
<td>35</td>
<td>File.Versions</td>
</tr>
<tr>
<td>35.1</td>
<td>Read the latest version</td>
</tr>
<tr>
<td>35.2</td>
<td>Date of the latest version</td>
</tr>
<tr>
<td>35.3</td>
<td>Write the next version</td>
</tr>
<tr>
<td>35.4</td>
<td>Purge old versions</td>
</tr>
<tr>
<td>35.5</td>
<td>Remove all versions</td>
</tr>
<tr>
<td>35.6</td>
<td>Get all versions</td>
</tr>
<tr>
<td>35.7</td>
<td>Read and write file bottlenecks</td>
</tr>
<tr>
<td>35.8</td>
<td>Creating and removing directories</td>
</tr>
<tr>
<td>36</td>
<td>CGI</td>
</tr>
<tr>
<td>36.1</td>
<td>Mime header</td>
</tr>
<tr>
<td>36.2</td>
<td>Document type</td>
</tr>
<tr>
<td>36.3</td>
<td>Special character encoding</td>
</tr>
<tr>
<td>36.4</td>
<td>HTML elements (generic)</td>
</tr>
<tr>
<td>36.5</td>
<td>HTML elements (specific shortcuts)</td>
</tr>
<tr>
<td>36.6</td>
<td>Standards</td>
</tr>
<tr>
<td>36.7</td>
<td>CGI inputs</td>
</tr>
<tr>
<td>37</td>
<td>Daytime</td>
</tr>
<tr>
<td>37.1</td>
<td>Data types</td>
</tr>
<tr>
<td>37.2</td>
<td>Lexing</td>
</tr>
<tr>
<td>37.3</td>
<td>Parsing</td>
</tr>
<tr>
<td>37.4</td>
<td>Instance declarations</td>
</tr>
<tr>
<td>37.4.1</td>
<td>Showing</td>
</tr>
<tr>
<td>37.4.2</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>37.5</td>
<td>Weekday methods</td>
</tr>
<tr>
<td>37.6</td>
<td>Daytime methods</td>
</tr>
<tr>
<td>38</td>
<td>HaskellLexer</td>
</tr>
<tr>
<td>38.1</td>
<td>Handling literate scripts</td>
</tr>
<tr>
<td>38.2</td>
<td>Lexing scripts</td>
</tr>
<tr>
<td>38.3</td>
<td>Handling the offside rule</td>
</tr>
<tr>
<td>38.4</td>
<td>Diagnostics</td>
</tr>
<tr>
<td>38.5</td>
<td>Poor man’s parsing</td>
</tr>
<tr>
<td>39</td>
<td>Playing Cards</td>
</tr>
<tr>
<td>39.1</td>
<td>Data types</td>
</tr>
<tr>
<td>39.2</td>
<td>Creating decks</td>
</tr>
<tr>
<td>39.3</td>
<td>Instance declarations</td>
</tr>
<tr>
<td>40</td>
<td>Poker</td>
</tr>
<tr>
<td>40.1</td>
<td>Presorting and grouping</td>
</tr>
<tr>
<td>40.2</td>
<td>Categorisation of hands</td>
</tr>
<tr>
<td>40.3</td>
<td>Comparison of hands</td>
</tr>
<tr>
<td>40.4</td>
<td>Hands with more than 5 cards</td>
</tr>
<tr>
<td>41</td>
<td>SendMail</td>
</tr>
<tr>
<td>41.1</td>
<td>Function</td>
</tr>
<tr>
<td>42</td>
<td>(Experimental) Data.Graph</td>
</tr>
<tr>
<td>42.1</td>
<td>Graph abstract data type</td>
</tr>
<tr>
<td>42.2</td>
<td>Sparse graph type</td>
</tr>
<tr>
<td>42.3</td>
<td>Graph Operations</td>
</tr>
<tr>
<td>42.4</td>
<td>Reachability</td>
</tr>
<tr>
<td>42.5</td>
<td>Cycles detection</td>
</tr>
<tr>
<td>43</td>
<td>(Experimental) MySQL C API Binding</td>
</tr>
<tr>
<td>43.1</td>
<td>API Data types</td>
</tr>
<tr>
<td>43.1.1</td>
<td>Basics</td>
</tr>
<tr>
<td>43.1.2</td>
<td>Connections</td>
</tr>
<tr>
<td>43.1.3</td>
<td>Results</td>
</tr>
<tr>
<td>43.1.4</td>
<td>Rows</td>
</tr>
<tr>
<td>43.1.5</td>
<td>Fields</td>
</tr>
<tr>
<td>43.1.6</td>
<td>Options</td>
</tr>
<tr>
<td>43.2</td>
<td>API Functions</td>
</tr>
</tbody>
</table>

44 (Experimental) MySQL Haskell API 32

44.1 Data types 32

44.1.1 Connections 32

44.2 Functions 32

44.2.1 Establishing a connection 32

44.2.2 Closing a connection 32

44.2.3 Issuing a query 32

44.2.4 Fetching query results 32

45 (Experimental) Database.Relational 33

45.1 Definitions and data types 33

45.2 Relational algebra 33

45.2.1 Projection of a tuple on a relation schema 33

45.2.2 Projection of a relation on a relation schema 33

45.2.3 Natural join 33

45.2.4 Union 33

45.2.5 Difference 33

45.2.6 Selection 33

45.2.7 Renaming 34

45.3 Datum operations 34

46 (Deprecated) Control.List 34

46.1 Backwards map 34

47 (Deprecated) Data.BSTree 34

47.1 BSTree type 34

47.2 BSTree operations 34

48 (Deprecated) Data.HashTables 34

48.1 Data types 34

48.2 Creating a new hash table 35

48.3 Updating an existing hash table 35

48.4 Looking up in a hash table 35

48.5 Dumping a hash table 35

49 (Deprecated) Data.Queue 35

49.1 Data type 35

49.2 Operations 35

50 (Deprecated) Data.SparseSet 35

50.1 Data type 35

50.2 Operations 35

51 (Deprecated) Data.Set 36

51.1 Data type 36

51.2 Operations 36

51.3 Instances 36

51.3.1 Ord 36

51.3.2 Showing 36

51.3.3 DeepSeq 36

52 (Deprecated) DeepSeq 36

52.1 Class Definition 36

52.2 Index operator 36

52.3 Instance Declarations 36

52.3.1 Simple instances 36

52.3.2 Tuple instances 37

52.3.3 List instance 37

52.3.4 Maybe instance 37

52.3.5 Either instance 37

53 (Deprecated) DeepSeq.BStree 37

53.1 Instance declaration 37

54 (Deprecated) Logic.Qualification 37

54.1 Qualifiable class 37

54.2 Instance declarations 37

54.2.1 Qualification 37

References 37

Index 37
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), Desmos (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+LPX source code. The symbol $ appears is 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 LPX documents.

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

2 Installation

2.1 Dependencies

ABR.Util.Args uses the Glob package. Install that on your system with cabal:

```
cabal install Glob
```

To compile where Glob is not available use the ghc compiler option -DNO_GLOB.

2.2 Building

This library is packaged for distribution in the file ABRHLibs.zip, available from http://http://www.ict.griffith.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.

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 LPX and Haskell sources, use

```
$ make CLEAN
```

To rebuild the file ABRHLibs.zip, use

```
$ make distribute
```

3 Util.Args

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

```
{-# LANGUAGE CPP #-}
module ABR.Util.Args (  
  OptSpec(..), OptVal(..), Options, findOpts, 
  assertFlagPlus, assertFlagMinus, deleteFlag, 
  insertParam, deleteParam, emptyOptions, 
  lookupFlag, lookupParam, lookupQueue  
) where

# ifdef NO_GLOB
  -- legacy for hobbit
# else
  , glob
# endif

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 Flag name. Users set or unset with +name or name respectively.
- a parameter with a value. Specify with Param 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 Queue 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 Param 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 | ParamVal String | ParamMissingValue | ParamQueue (S.Seq String)  
  deriving Show

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

type Options = M.Map String OptVal

3.2 Empty options

emptyOptions is an empty Options.

emptyOptions :: Options

3.3 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 mapping from the option names to the values found; and leftovers is a list of any unconsumed arguments, typically file names.

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

3.4 Adding and deleting options

Flags can be asserted positive or negative or deleted, with assertFlagPlus, assertFlagMinus, and deleteFlag, respectively.

```
assertFlagPlus, assertFlagMinus, deleteFlag ::  
  String -> Options -> Options
```

September 12, 2018
Params can be inserted or deleted, with `insertParam` and `deleteParam`, respectively.

```haskell
insertParam :: String -> String -> Options -> Options
deleteParam :: String -> Options -> Options
```

### 3.5 Looking up options

- **lookupFlag**
  - `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.
  ```haskell
  lookupFlag :: String -> Options -> Bool -> Bool
  
  lookupParam
  ```
  - `lookupParam name options def` returns the value stored for the `name`ed parameter in `options` or `def` if it has not been properly specified.
  ```haskell
  lookupParam :: String -> Options -> String -> String
  
  lookupQueue
  ```
  - `lookupQueue name options` returns the list stored for the `name`ed queue parameter in `options` or `[]` if it has not been properly specified.
  ```haskell
  lookupQueue :: String -> Options -> [String]
  ```

### 3.6 Filename globbing

- **glob**
  - `glob paths` returns the really existing file names that match the `paths` which may contain wildcards.
  ```haskell
  # ifdef NO_GLOB
  -- legacy for hobbit
  # else
  glob :: [FilePath] -> IO [FilePath]
  # endif
  ```

## 4 Util.Errors

Module `ABR.Util.Errors` provides a framework for collecting warnings and errors, displaying them, and exiting on fatal errors.

```haskell
module ABR.Util.Errors (WEMessage(..), WEResult(..),
returnPass, returnFail, returnWarn) where

4.1 Warning and error data type

A `WEMessage` always has:

- a category (`weCat`);
- the text of the message (`weMsg`);

and may have:

- a position `wePos`.

The category is a type provided by the application.

```haskell
data WEMessage cat = WEMessage {
  weCat :: cat,
  wePos :: Maybe Pos,
  weMsg :: String
}
```

4.2 Warning and error category type class

Class `ErrorCategory` includes all types that categorise errors or warnings.

```haskell
class ErrorCategory cat where
isFatal x returns true if x denotes a fatal, unrecoverable error.

isFatal :: cat -> Bool
```

### 4.3 Data types

A phase of a compiler will typically return a list of `WEMessages`, along with the result of the phase if there is one. A `WEResult` distinguishes the cases that have passed (for example with only warnings) from those that have failed (with a fatal error). `a` is the phase’s result.

```haskell
data WEResult cat a = WEPass [WEMessage cat] a |
  WEFail [WEMessage cat]
```

### 4.4 Returning successfully, failing or warning

- **returnPass** `x` returns `x` as a passing value.
- **returnFail** returns a failing message, `wem`.
- **returnWarn** `wem` returns a warning message, `wem`.

```haskell
returnPass :: a -> WEResult cat a
returnFail :: WEMessage cat -> WEResult cat ()
returnWarn :: WEMessage cat -> a -> WEResult cat a
```

## 5 Util.Open

Module `ABR.Util.Open` allows access to system facilities to open things like URLs.

```haskell
module ABR.Util.Open (openURL) where

5.1 Opening a URL

- **openURL** `url` opens the `url` in the default browser.

```haskell
openURL :: String -> IO ExitCode
```

## 6 Util.Pos

Module `ABR.Util.Pos` defines a type for a position in a source code.

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

6.1 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”.

```haskell
type Line = Int
type Col = Int
type Pos = (Line, Col)
```
6.2 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`.

```hs
class HasPos a where
    getPos :: a -> Pos

getPos x = error "undefined HasPos instance"
```

6.2.1 Container instances

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

6.3 Relative positions

A position `p1` **precedes** another position `p2` if `p1` comes earlier in than `p2`.

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

7 Util.Time

Module `ABR.Util.Time` collects time-related functions. The new time libraries are complicated. So are the old ones. The transition is not smooth. GHC 7.0.x (hobbit) is not compatible with the current versions, hence all the legacy stuff implemented with conditional compilation.

```hs
module ABR.Util.Time (LegacyTime, C.UTCTime, LT.ZonedTime, LT.LocalTime,
    utcToZonedTime, utcToLocalTime, getCurrentUTCTime, getCurrentLocalTime,
    getCurrentZonedTime, TimeFormat, dateThenTime1, dateThenTime2,
    formatTime, formatUTCTime, currentTime, fileModTime,
    LegacyTimes (getCurrentLegacyTime, diffSec, diffMin, diffHour, diffDay,
    formatLegacyTime)) where
```

7.1 Data types

`LegacyTime` is a type synonym for either `ClockTime` or `UTCTime`, whichever is returned by `getModificationTime`.

```hs
type LegacyTime =
```

7.2 Time conversions

- `utcToZonedTime` converts a UTC time to a ZonedTime.
- `utcToLocalTime` converts a UTC time to a LocalTime.

```hs
utcToZonedTime :: C.UTCTime -> IO LT.ZonedTime
utcToLocalTime :: C.UTCTime -> IO LT.LocalTime
```

7.3 Getting the current time

- `getCurrentUTCTime` gets the current time in UTC.
- `getCurrentLocalTime` gets the current local time.

```hs
getCurrentUTCTime :: IO C.UTCTime
getCurrentLocalTime :: IO LT.LocalTime
```

7.4 Format strings

A `TimeFormat` is a time format string.

```hs
type TimeFormat = String
```

These are strings to use with the time `formatTime`.

- `format` produces `dateThenTime1` Wed 27 Nov 2013 10:16:14 EST
- `format` produces `dateThenTime2` Wed 27 Nov 2013 10:16:14

```hs
dateThenTime1, dateThenTime2 :: TimeFormat
```

7.5 Formatting times

- `formatUTCTime` returns a formatted UTC time in local time showing the time zone.

```hs
formatUTCTime :: C.UTCTime -> IO String
```

7.6 Formatted current time

- `currentTime` returns a the formatted current time in local time showing the time zone.

```hs
currentTime :: IO String
```

7.7 Formatted file modification time

- `fileModTime` returns the formatted modification time of a file.

```hs
fileModTime :: FilePath -> IO String
```

7.8 Overloading for old/new time systems

Class `LegacyTimes` overloads time operations with old and new time types.

```hs
class LegacyTimes t where
    getCurrentLegacyTime :: IO t
    t1 `diffSec` t2 returns `t1` − `t2` rounded down (floor) to whole seconds.
    diffSec :: t -> t -> Int
    t1 `diffMin` t2 returns `t1` − `t2` rounded down to whole minutes.
    diffMin :: t -> t -> Int
    t1 `diffHour` t2 returns `t1` − `t2` rounded down to whole hours.
    diffHour :: t -> t -> Int
    t1 `diffDay` t2 returns `t1` − `t2` rounded down to whole days.
    diffDay :: t -> t -> Int
```

7.9 Instances

7.9.1 LegacyTimes

```hs
instance LegacyTimes C.UTCTime where
    # if __GLASGOW_HASKELL__ < 705
    getCurrentLegacyTime :: IO t
    t1 `diffSec` t2 returns `t1` − `t2` rounded down (floor) to whole seconds.
    diffSec :: t -> t -> Int
    t1 `diffMin` t2 returns `t1` − `t2` rounded down to whole minutes.
    diffMin :: t -> t -> Int
    t1 `diffHour` t2 returns `t1` − `t2` rounded down to whole hours.
    diffHour :: t -> t -> Int
    t1 `diffDay` t2 returns `t1` − `t2` rounded down to whole days.
    diffDay :: t -> t -> Int
```

```hs
# else
# endif
```
8 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.\footnote{Thanks to Daniel Young for suggested extensions to this module.}

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

8.1 Data type

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

```haskell
data CheckResult passType failType =  
  CheckPass passType  
  | CheckFail failType
```

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

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

8.2 Sequencing checks

```haskell
(-# DEPRECATED msort "Use Data.List.sort" -)
msort :: (a -> a -> Bool) -> [a] -> [a]
```

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

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

`sorbYLength` sorts a list of lists into non-descending order of length.

9 Control.Map


module ABR.Control.Map (lookupGuard) where

import qualified Data.Map as M

9.1 lookupGuard

```haskell
lookupGuard m 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.
```

```haskell
lookupGuard :: Ord a => M.Map a b -> [a] -> (a -> c) -> ([b] -> c) -> c
```

10 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_ge1',  
    properSublists, pPlus, meet, disjoint, allUnique,  
    duplicates, mub, isSubset, findSubset, noSuperSets,  
    isSubSequence, notSubSequence, chop,  
    dropsuffix,掉了, oseft, oseftion,  
    sortByLength, trimN, trim2, replace1, replaceAll  
) where

10.1 Sorting

```haskell
msort :: lt xs sorts xs using lt as the less-than operator.
```

```haskell
sortByLength :: [x] -> [x]
```

10.2 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.

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

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

`cartProd` produces the cartesian product of an arbitrary number of lists. That is, `cartProd [xs1, xs2, ...]` returns `xs1 × xs2 × ...`. Note: Prelude.sequence can be used to do the same job.

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

```haskell
interleave [] xs returns the list of lists formed by inserting `x` in each possible place in `xs`.
```

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

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

`separate` concats `yss` with `zs` interspersed.

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

fragments :: [a] 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`.

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

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

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

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

`permutations` `k` `xs` returns all the permutations of `k` elements selected from `xs`. Precondition: `0 <= k <= length xs`. 

September 12, 2018 7
permutations :: Int -> [a] -> [[a]]
{# DEPRECATED permutations "Use Data.List.permutations" #}
combinations k xs returns all the combinations of k elements drawn from xs.
Precondition: 0 ≤ k ≤ 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]]
chnopSet_ge1 :: [a] -> [[a]]
chnopSet_ge1' :: [a] -> [[a]]
xss

10.4 Set-like operations

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

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

10.5 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 x n), for m = length ps and n = 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 suf and sep are appended. Use this to create output file names from input file names.
subsSuffix :: Eq a => a -> [a] -> [a]

trim\n n xs drops n elements from both ends of a list.
trim\n :: Int -> [a] -> [a]

trim2 xs drops 2 elements from both ends of a list. Useful for comments and the like.
trim2 :: [a] -> [a]

replace1 as bs xs replaces the first occurrence of as in xs with bs.
replace1 :: Eq a => [a] -> [a] -> [a] -> [a]

replaceAll bs xs replaces all occurrences of as in xs with bs.
replaceAll :: Eq a => [a] -> [a] -> [a] -> [a]
Module `ABR.Data.NameTable` implements structures for the efficient accumulation of names, assigning unique, sequential integers to each name and mapping between them.

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

11.1 Data types

A NameTable is a pair of:

- a map from strings to integers; and
- a supply of sequential integers starting from 0.

type NameTable =

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

type NameArray = Array Int String

11.2 Creating a name table

newNameTable size creates a NameTable.

```haskell
newNameTable :: 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.

```haskell
insertName :: NameTable -> String -> IO NameTable
```

11.3 Looking up by name

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

```haskell
lookupName :: NameTable -> String -> Maybe Int
```

11.4 Creating a name array

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

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

12 Data.Supply

Module `ABR.Data.Supply` implements a name supply using a mutable variable in the IO monad.

```haskell
module ABR.Data.Supply (Supply, newSupply, supplyNext, peekNext) where

12.1 Data types

A Supply is a value of any enumerated type.

type Supply a =

12.2 Creating a supply

newSupply first creates a new Supply that will commence with first.

```haskell
newSupply :: a -> IO (Supply a)
```

12.3 Extracting values from a supply

supplyNext supply returns the next value from the supply.

```haskell
supplyNext :: Exum 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.

```haskell
peekNext :: Supply a -> IO a
```

13 Debug.Array

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

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

13.1 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.

```haskell
{(!!!)} :: (Ix i, Show i, Show e) => Array i e -> 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.

```haskell
array' :: (Ix i, Show i, Show e) => Array i 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.

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

14 Debug.IArray

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

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

14.1 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.

```haskell
{(!!!)} :: (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.

```haskell
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.

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

15 Parser

The `ABR.Parser` module provides a framework for lexical analysis and parsing using parser combinators [1, 2].

```haskell
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

15.1 Error messages

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

type Msg = String
15.2 Results

An analyser could succeed, fail or generate an error. The `Could`, `OK`, or `Error` type wraps around any other type to indicate success (with 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)

15.3 Analysers

An `Analyzer` 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 Analyzer a b = [(a,Pos)] -> Could (b, [(a,Pos)])

15.4 Elementary analysers

This is the simplest analyser. `succeed` succeeds with a predefined value and does not consume any input.

`succeed` :: b -> Analyzer a b

`epsilon` is the trivial case of `succeed`. It always succeeds and returns the trivial value `()`.

`epsilon` :: Analyzer a ()

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

`fail` :: Msg -> Analyzer a b

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

`error` :: Msg -> Analyzer a b

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

`end` :: Analyzer a ()

15.5 Elementary analyser combinators

These combinators allow the composition of analysers.

`(<|>)` is the alternation combinator. `a1 <|> a2` returns the result of `a1`, or `a2` if `a1` failed.

`(<|>)` :: Analyzer a b -> Analyzer a c -> Analyzer a b

`(&)` is the sequence combinator. `a1 & a2` returns a pair formed from the results of `a1` and `a2`.

`(&)` :: Analyzer a b -> Analyzer a c -> Analyzer a (b,c)

15.6 Analyser result modifiers

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

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

`(&)` :: Analyzer a b -> (b -> c) -> Analyzer a c

`a . f` changes the value returned by analyser `a` by replacing it with `f`.

`( . f)` :: Analyzer a b -> c -> Analyzer a c

15.7 More analyser combinators

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

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

`some` changes analyser `a` which recognizes one thing into an analyser which recognizes a sequence of one or more things, returned in a list.

`many` changes analyser `a` which recognizes one thing into an analyser which recognizes a sequence of zero or more things, returned in a list.

`optional` 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` :: Analyzer a b -> Analyzer a [b]

`someUntil` `a1 a2` creates an analyser that recognizes a sequence of one or more of the things recognized by analyser `a1`, like `some`, but stops consuming input when a second analyser `a2` would also work.

`manyUntil` `a1 a2` creates an analyser that recognizes a sequence of zero or more of the things recognized by analyser `a1`, like `many`, but stops consuming input when a second analyser `a2` would also work.

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

15.8 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` :: String -> [(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

September 12, 2018 10

Thanks to Chris English for suggesting this strategy.
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.

```haskell
data 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.

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

### 15.9 Elementary lexers

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

This is a common lexical structure for sources:

```
source ::= "|.lexeme1| |lexeme2| |...|
```

**literalL** :: Char -> Lexer

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

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

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

15.12 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 starts with a capital P.

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

### 15.13 Elementary parsers

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

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

**literalP** :: Lexeme -> Tag -> Lexer

```haskell
literalP :: Lexeme -> Tag -> Lexer
```

15.14 Parser result modifiers

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

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

15.15 Error reporting

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

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

15.16 Instance declarations

instance HasPos (Could a) where

### 16 Parser.Lexers

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

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

### 16.1 Frequently used lexers


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

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

```haskell
space := "\t".
```

```haskell
source
```

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

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

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

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".

signedCardinalL recognizes a signed whole number.

signedCardinal ::= ["-"] cardinal.

signedFixedL recognizes a signed fixed number.

signedFixed ::= ["-"] fixed.

signedFloatL recognizes a signed floating point number.

signedFloat ::= ["-"] float.

17 Parser.Checks

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

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

17.1 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 :: Lexer -> Parser a -> Check String a String

18 Parser.Predicates

The ABR.Parser.Predicates module provides some frequently used predicates that depend on lexing/parsing.

module ABR.Parser.Predicates (isCardinal, isFixed, isFloat, isSignedCardinal, isSignedFixed, isSignedFloat) where

18.1 isNumber predicates

isCardinal, isFixed, isFloat, isSignedCardinal, isSignedFixed, isSignedFloat :: String -> Bool
19 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 or JSON, but this format is nicer to edit by hand. Its design was inspired by the configuration files for Apache.

module ABR.Text.Configs (Config(..), Configs, configsL, configsP, stringL, showConfigs, read', lookupConfig, updateConfig, lookupFlag, lookupParam, getParam, getValueList, popTemplate) where

19.1 Data types

A configuration, Config, is one of:

- CFlag a flag that is set by the presence of its name, cName, (or just an item of data);
- CParam a parameter with name, cName, and an associated value, cValue;
- CSet a parameter with a with name, cName, and an associated set of configurations, cSet; or
- CList a parameter with a with name, cName, and an associated list of configurations, cList.

data Config = CFlag {cName :: String} |
| CParam {cName :: String, cValue :: String} |
| CSet {cName :: String, cSet :: Configs} |
| CList {cName :: String, cList :: [Configs]} |

A Configs is a list of configurations.

type Configs = [Config]

A config file will normally contain a Configs, that is a sequence of Configs not separated by commas. For example, this is just a sequence of flags, turned on or off by commenting them out.

```
# flags.config
verbose
optimise
background
```

The lookupConfig function, below, implements a query language for Configs. For the # flags.config example, lookupConfig with query "verbose" would return Just (CFlag "verbose"), and Nothing for "optimise".

For these kind of flags, use the function lookupFlag which will return a more convenient Bool.

Another type of Config is a parameter. A parameter has a name and an associated value.

```
# param.config
verbose # a flag
level = 3 # a parameter (name = level, value = 3)
name = "System X"
```

The name permits the value for a parameter to be looked up with a query. lookupConfig with query "level" would return Just (CParam "level" "3"). The more convenient function lookupParam would return Just "3".

Where related Configs may be logically grouped, use a set. Sets have names like parameters and the values are Configs delimited by a set of braces.

```
# set.config
runOptions = {
  verbose # a flag
  # optimise
}
site = {
  level = 3 # a parameter (name = level, value = 3)
  name = "System X"
}
```

Access members of a set with a period. Example query: "site.level".

Where there will be multiple instances of Configs with all of the same Configs, use a named list.

```
# list.config
runOptions = {
  verbose # a flag
  # optimise
}
site = {
  level = 3 # a parameter (name = level, value = 3)
  name = "System X"
  courses = [
    # this site manages multiple courses
    code = "1806ICT"
    ,
    code = "1801ICT"
    ,
    code = "1701ICT"
    ,
  ]
  name = "Creative Coding"
}
```

Access elements of a list with the indexing operator !. For example lookupConfig "site.courses" returns Just the whole CList. lookupParam "site.courses!1.code" would return Just "1801ICT". Note that that all queries must end in a name, and not with indexing !n, as the elements of a list are Configs, not Configs. Remember that a Configs is a synonym for [Config].

Note that the elements of a list are Configs, and therefore may be a sequence of Configs, not separated by commas. Commas are only used to separate the list elements.

A list might contain just a sequence of strings, numbers, or flags, all of which will be parsed as flags.

```
# values.config
runOptions = {
  verbose # a flag
  # optimise
}
site = {
  level = 3 # a parameter (name = level, value = 3)
  name = "System X"
  courses = [
    # this site manages multiple courses
    code = "1806ICT"
    ,
    code = "1801ICT"
    ,
    code = "1701ICT"
    ,
  ]
  name = "Creative Coding"
  notifyEmails = [
    a.rock@griffith.edu.au,
    j.faichney@griffith.edu.au,
  ]
}
```

September 12, 2018
19.2 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.

```plaintext
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, at-signs, and slashes. Note that a number can lex as a name, as could many file paths. Names are case sensitive.

```plaintext
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.

```plaintext
string ::= "\n" {"\n" | $anything not \""} "\n";
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.

```plaintext
configsL ::= {comment | name | string | "=" | "{" | "}" | "[" | "]" | "," | $whitespace$};
level="lexical".
```

19.3 Parser
A value is either a name or a string.

```plaintext
value ::= name | string;
level="grammar".
```

A config is a sequence of whitespace separated configs, parsed by `configsP`.

```plaintext
configs ::= {config};
level="grammar".
```

A configSet is a configs in braces.

```plaintext
configSet ::= { configs };
level="grammar".
```

A configList is a comma separated sequence of configs in brackets.

```plaintext
configList ::= [ configs ];
level="grammar".
```

A config is either: a binding of a name to a configSet, a configList, or a value; or just a name.

```plaintext
config ::= name "=" value
 | name "=" configSet
 | name "=" configList
 | value;
level="grammar".
```
19.4 Showing

instance Show Config where

    showConfigs cs shows a list of configs.

showConfigs :: Configs -> String

19.5 Reading

read' s may be used to read a parameter value s, removing quotes first.

read' :: Read a => String -> a

19.6 Accessing

19.6.1 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.

19.6.2 Lookup functions

lookupConfig n cs tries to find the named config or configs n in cs, returning Just c if the first match is a config, or Nothing. n is a configPath.

updateConfig :: String -> Configs -> Maybe Config

updateConfig n 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.

lookupFlag n cs tries to find the named flag n in cs, returning True iff it exists as a flag. n is a configPath.

lookupFlag :: String -> Config -> Configs -> Configs

lookupParam n cs tries to find the named parameter n in cs, returning Just the first value or Nothing.

lookupParam :: String -> Config -> Configs -> Maybe String

getParam n cs tries to return the value for the named CParam. It is an error if the parameter cannot be found.

getParam :: String -> Configs -> String

getValueList n cs tries to find the named CList and return the list of single values in it. It is an error if the list can not be found or if any of the elements of the list are not a single CFlag.

getValueList :: String -> Configs -> [String]

19.7 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.

19.7.1 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 it 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:
  - _UNDEFINED_, if the Config does not exist;
  - the unStringed text of the flag 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.
- In the following sequences, no extra whitespace is permitted.
- The sequence #ifdef#configPath#text#ENDIF# 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 #ifndef#configPath#text#ENDIF# 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 #ifdefined#configPath#text#ENDIF# 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 lead to a Config that exists, but is not a set, the output is _NOT_A_SET_.
- The sequence #for each#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_.

19.7.2 Populating templates

popTemplate config 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

20 Text.JSON

The ABR.Text.JSON library provides functions to parse, construct, and interrogate JSON data.

module ABR.Text.JSON (  
    JPropertyName, JValue(..),  
    jmString, jmNumber, jmTrue, jmFalse, jmNull,  
    jmArray, jmObject,  
    lexerL, valueP,  
    JQuery (..), JResult (..)  
) where

20.1 Basic data types

A distinction must be made between string data that appears in the JSON text (delimited with double quotes, and with special character escaping) and string data that is ready to use in an application (without all that). In this document, escaping is the process of adding quotes and escaping special characters, and desescaping is the reverse process.

A JPropertyName was lexed as a JSON string and has been desescaped.

type JPropertyName = String  
A JValue is a lump of JSON data.

data JValue =
20.2 Construction

jmString cs constructs a JSON string value containing cs.

jmString :: String -> JValue

jmNumber x constructs a JSON number value containing x.

jmNumber :: (Num a, Show a) => a -> JValue

jmTrue constructs the JSON value true. jmFalse constructs the JSON value false. jmNull constructs the JSON value null.

jmTrue, jmFalse, jmNull :: JValue

jmArray vs constructs a JSON array from vs.

jmArray :: [JValue] -> JValue

jmObject nvs constructs a JSON object from nvs, where nvs is a list of pairs, (n,v), n is a property name, and v is its value.

jmObject :: [(JPropertyName, JValue)] -> JValue

20.3 Lexer

lexerL lexes a JSON source.

lexerL :: Lexer

20.4 Parser

valueP parses a JSON source.

valueP :: Parser JValue

20.5 Interrogation

A JQuery is a request to project out a part of a JValue. It has the following values:

- JQStr - The value is a string, return it as a String.
- JQInt - The value is a number, return it as an Int.
- JQDbe - The value is a number, return it as a Double.
- JQBool - The value is true or false, return it as a Bool.
- JQProp n - The value is an object, return the property named n.
- JQProps - The value is an object, return all of the the properties as a list of (name, value) pairs.
- JQElem i - The value is an array, return the ith element.
- JQElems - The value is an array, return all the elements, in order, in a list; or the value is an object, return all the property values.
- JQIsNull - The value might be null, return True if it is.
- q1 :-> q2 - This query is really a sequence of two queries to be applied in the order q1 then q2.

infixl 9 :->

data JQuery =
| JQStr |
| JQInt |
| JQDbe |
| JQBool |
| JQProp JPropertyName |
| JQProps |
| JQElem Int |
| JQElems |
| JQIsNull |
| JQuery :-> JQuery

Class JResult overloads jGet.

jGet q v applies the query q to value v. It may fail with Nothing.

class JResult a where
jGet :: JQuery -> JValue -> Maybe a

20.6 Instances

20.6.1 Showing

The Show instances reconstruct valid JSON syntax.

instance Show JValue where

20.6.2 Interrogation

instance JResult String where
instance JResult Int where
instance JResult Double where
instance JResult Bool where
instance JResult JValue where
instance JResult [JValue] where

21 Text.CSV

The ABR.Text.CSV library provides functions to parse, construct, and interrogate CSV data.

module ABR.Text.CSV where

21.1 Basic data types

A CSV file is one big string.

type CSV = String

A Field is one cell.

type Field = String

A Row is one list of Fields.

type Row = [Field]

A Table is a just list of Rows.

type Table = [Row]

21.2 Parser

Don’t really need a full-on parser.

21.3 Interrogation

21.4 Instances

21.4.1 Showing

The Show instances reconstruct valid CSV syntax.

21.4.2 Interrogation

22 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

22.1 Making text safe for HTML

encodeHTML c returns c’s special character encoding if c ∈ {<, >, &, "}, otherwise c.

encodeHTML’ :: Char -> String

makeHTMLSafe cs encodes all of the special characters in cs. It would be counterproductive to put text containing tags through this filter. makeHTMLSafe’ :: String -> String

makeHTMLSafe, makeHTMLSafe’ :: String -> String
22.2 Making text safe for LaTeX

\[\text{makeLatexSafe } cs \text{ makes } cs \text{ safe for inclusion in a \LaTeX\ document as plain text, by encoding some special characters.}\]

makeLatexSafe :: String -> String

22.3 Converting LaTeX to HTML

\[\text{latex2html } cs \text{ converts \LaTeX\ string } cs \text{ to HTML. This is not meant for whole documents. It has some basics for writing comments just like this one. This is used by mashdoc.}\]

latex2html :: String -> String

23 Text.String

Module ABR.Text.String

\text{is a collection of functions that operate on strings.}

module ABR.Text.String (wordWrap, lJustify, rJustify, lJustify', rJustify', justifyColumn, makeTable, spaceColumns, makeTableMR, fields, unfields, trim, nameCmp, nameLT, fixNewlines, fixNewlines', spaces, findClosest, (+/=+), (+++,++), catenateWith, substs, subst, subHashNames, subHashNums, unString, enString) where

23.1 Word wrapping

\[\text{wordWrap width } cs \text{ wraps the words in } cs \text{ to no wider than width, unless a word is wider than width, returning a list of lines.}\]

wordWrap :: Int -> String -> [String]

23.2 Justification

\[\text{lJustify } w \text{ pads } cs \text{ with extra spaces on the right to make the overall width not less than } w.\]

lJustify :: Int -> String -> String

\[\text{rJustify } w \text{ pads } cs \text{ with extra spaces on the left to make the overall width not less than } w.\]

rJustify :: Int -> String -> String

\[\text{lJustify' } p \text{ pads } cs \text{ with extra pad characters } p \text{ on the right to make the overall width not less than } w.\]

lJustify' :: Int -> String -> String

\[\text{rJustify' } p \text{ pads } cs \text{ with extra pad characters } p \text{ on the left to make the overall width not less than } w.\]

rJustify' :: Int -> String -> String

23.3 Tables with justified columns

\[\text{justifyColumn j col justifies all of the strings in } col \text{ using } j \text{ to justify them all to the same width, which is the widest width in } col.}\]

justifyColumn :: (Int -> String -> String) -> [String] -> [String]

\[\text{makeTable } js \text{ applies the justification functions in } js \text{ to the corresponding columns in } cols \text{ and assembles the final table. Short columns have extra blank rows added at the bottom.}\]

makeTable :: [Int -> String -> String] -> [String] -> String

\[\text{spaceColumns } cs \text{ spaces out columns } cols \text{ by inserting columns of replicated strings } cs.}\]

spaceColumns :: String -> [String] -> [String]

23.4 Fields

\[\text{These are functions for breaking a string into a list of fields and converting a list of fields into a string. The fields are delimited with a nominated special character. To permitt the special character to appear in a field it is preceded by a nominated escape character. To permit the escape character to appear in a string, it is preceded by itself.}\]

fields :: (Char -> String -> String) -> [String] -> String

\[\text{unfields } d e \text{ breaks string } cs \text{ into a list of strings at each delimited character } d, \text{ removing escape characters } e \text{ where appropriate. If the escaping is not required use ABR.Data.List.intersperse instead.}\]

unfields :: (Char -> String -> String) -> [String] -> String

23.5 Whitespace

\[\text{trim } cs \text{ strips any whitespace from both ends of } cs.}\]

trim :: [String] -> String

\[\text{fixNewlines } cs \text{ rectifies the ends of lines in } cs. \text{ It does not ensure that the last character is a newline.}\]

fixNewlines :: String -> String

\[\text{spaces' } n \text{ returns } n \text{ spaces.}\]

spaces :: Int -> String

23.6 Pattern matching and substitution

\[\text{findClosest candidates returns the position in candidates of the string which, ignoring case is closest to pattern or } -1 \text{ if candidates is empty.}\]

findClosest :: [String] -> String -> Int

\[\text{substs } p s r \text{ performs substitutions on } cs. \text{ } prs \text{ is a list of pairs } (p, r), \text{ where } p \text{ is a case sensitive pattern to be replaced by } r \text{ wherever it occurs.}\]

substs :: ([String, String]) -> String -> String

\[\text{subst } p c r \text{ performs substitutions on } cs. \text{ } pc \text{ is a case sensitive pattern to be replaced by } r \text{ wherever it occurs.}\]

subst :: (String, String) -> String -> String

\[\text{subHashNums } rs \text{ performs substitutions on } cs. \text{ } rs \text{ is a list of replacements. } r!!0 \text{ will replace the pattern } #0, \text{ r!!1 will replace the pattern } #1, \text{ ...}\]

subHashNums :: [String] -> String -> String

\[\text{subHashNames } nr s \text{ performs substitutions on } cs. \text{ } nr \text{ is a list of pairs } (n, r), \text{ where } n \text{ is a case sensitive name to be replaced by } r \text{ wherever it occurs between a pair of hashes.}\]

subHashNames :: ([String, String]) -> String -> String

23.7 Names

\[\text{nameCmp } n1 n2 \text{ orders } n1 \text{ and } n2. \text{ Use this to sort names with sortBy when names are in family-name comma other-names format.}\]

nameCmp :: String -> String -> Ordering

\[\text{nameLT } n1 n2 \text{ returns True if name } n1 \text{ < } n2. \text{ Use this to sort names with msort when names are in family-name comma other-names format.}\]

nameLT :: String -> String -> Bool
23.8 Path catenation operators

`++/+` joins two paths with a single `/`, `++/++` joins two paths with a single `. The utility of these operators is that any extra `/` or `. at the join are removed.

```haskell
infixl 6 ++/+ , ++/++
```

More such operators can be constructed with

```haskell
catenateWith :: Char -> String -> String -> String
```

catenateWith :: Char -> String -> String -> String

23.9 Simple String Delimitation

```haskell
unString :: String -> String
```

`unString` 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.

```haskell
unString :: String -> String
```

```haskell
enString :: String -> String
```

`enString` encodes a string with enclosing quotes and doubles any enclosed quotes.

```haskell
enString :: String -> String
```

24 Text.Showing

The `ABR.Text.Showing` library provides functions to help write new instances of class `Show`, and to get control of numeric precision.

```haskell
module ABR.Text.Showing (
    showWithSep, showWithTerm, FormattedDouble(..),
    makeFormattedDouble, showFD
) where
```

24.1 Adding Delimiters

```haskell
showWithSep sep xs shows the elements of `xs` separated by `sep`.
```

```haskell
showWithTerm term xs shows the elements of `xs` terminated by terminator `term`. (Adapted from Mark Jones’s Mini Prolog.)
```

```haskell
showWithSep, showWithTerm ::
    Show a => String -> [a] -> ShowS
```

24.2 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.

```haskell
data FormattedDouble = FD (Maybe Int) Double |
    ED (Maybe Int) Double |
    GD (Maybe Int) Double
```

This `Show` instance applies the formatting to the `Double`.

```haskell
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 

- \(\text{“f”} \mid \text{“e”} \mid \text{“g”}\) (digits). e.g. \(\text{“f2”}\).

makeFormattedDouble format x = makeFormattedDouble from a Double x and a string that describes the format, format, of the form 

- \(\text{“f”} \mid \text{“e”} \mid \text{“g”}\) (digits). e.g. \(\text{“f2”}\).

25 Logic.Kinds

Module `Kinds` implements kinds for arguments in logic atoms.

```haskell
module ABR.Logic.Kinds (
    Kind(..), HasKind(..), kindCheckList, kUnify
) where
```

25.1 Data type

An `Kind` is the essential structure of the elements of a type. Only types of the same kind may be combined in certain ways, for example union. The traditional kind of type is the one that consist of named constants (`KNamed`). Types may also be subsets of the integers (`KIntegral`), or sets of strings (`KString`). Cartesian products of types yield tuple kinds (`KTuple`). There should never be less than two elements in a tuple kind. A with variables in it has an unknown kind (`KUnknown`).

```haskell
data Kind = KNamed | KIntegral | KString | KTuple [Kind] | KUnknown deriving (Eq, Ord)
```

25.2 Unification

```haskell
kUnify k k’ returns the unification of `k` and `k’` if there is one.
```

```haskell
kUnify :: Kind -> Kind -> Maybe Kind
```

25.3 Kind inference

Class `HasKind` overload functions pertaining to kinds.

class (Show a, HasPos a) => HasKind a where

```haskell
kindCheck :: a -> IO Kind
kindCheckList :: HasKind a => [a] -> IO Kind
```

25.4 Instance declarations

25.4.1 Showing

A named kind is represented as *. An integral kind is represented as $. A string kind is represented as $. Tuples are formed with parentheses and commas. An unknown kind (what a variable must have) is represented as ?.

```haskell
instance Show Kind where
    instance Show Kind where
    instance Show Kind where
```

26 Logic.Constamts

Module `Logic.Constants` implements constants.

```haskell
module ABR.Logic.Constamts (Constant(..), constantP, integerP, HasConstants(..)) where
```

26.1 Data types

An `Constant` is a fixed token which may appear as an argument to an atom. Constants may be the traditional named kind (`CNamed`), an integer (`CIntegral`), or a string (`CString`).

```haskell
data Constant =
    CNamed {
        cName :: String,
        cPos :: Pos
    } |
    CIntegral {
        cInt :: Integer,
        cPos :: Pos
    }
```
26.2 Parsers

integer ::= ["-" ] cardinal.

integer

integerP recognises integers, returning the integer value and the position it occurred at.

integerP : Parser (Integer, Pos)

constant ::= uName | integer | string.

constant

constantP recognises constants.

constantP : Parser Constant

26.3 Collecting constants

It is required for various purposes to identify all of the distinct constants that occur in an object. Constants can be collected from instances of class HasConstants.

class HasConstants a where

getConstants x C adds any constants in x to C.

getConstants :: a -> S.Set Constant -> S.Set Constant

hasConstants x returns True iff x contains constants.

hasConstants :: a -> Bool

26.4 Instance declarations

26.4.1 Comparing

instance Eq Constant where

instance Ord Constant where

26.4.2 Positions

instance HasPos Constant where

26.4.3 Showing

instance Show Constant where

26.4.4 Collecting constants

instance HasConstants Constant where

26.4.5 Kind inference

instance HasKind Constant where

26.4.6 DeepSeq

instance NFData Constant where

27 Logic.Variables

Module ABR.Logic.Variables implements variables.

module ABR.Logic.Variables ( Variable(..), variableP, HasVariables(..), Substitution(..), Groundable(..) ) where

27.1 Data type

An Variable is a token which may appear as an argument to an atom, to be instatiated with constants.

data Variable =

  Variable { vName :: String,
             vPos :: Pos }

27.2 Parsers

variable ::= lName.

variable

variableP recognises variables.

variableP : Parser Variable

27.3 Collecting variables

It is required for various purposes to identify all of the distinct variables that occur in an object. Variables can be collected from instances of class HasVariables.

class HasVariables a where

getVariables x V adds any variables in x to V.

getVariables :: a -> S.Set Variable -> S.Set Variable

hasVariables x returns True iff x contains variables.

hasVariables :: a -> Bool

27.4 Grounding

To ground is to substitute a variable with a constant.

A Substitution v :->- c replaces a variable v with a constant c. Substitutions may be composed. s1 :->- s2 first performs s1 and then s2. NullSub is the null substitution that does nothing.

data Substitution = NullSub

| Variable :->- Constant
| Substitution :-> Substitution

deriving (Eq, Ord, Show)

Anything groundable should be an instance of class Groundable.

class Groundable a where

groundi v c x returns x with all occurrences of variable v replaced by constant c.

groundi :: Variable -> Constant -> a -> a

ground s x applies substitution s to x.

ground :: Substitution -> a -> a

rename v v' x returns x with all occurrences of variable v replaced by another variable v'.

rename :: Variable -> Variable -> a -> a

27.5 Instance declarations

27.5.1 Comparing

instance Eq Variable where

instance Ord Variable where

27.5.2 Positions

instance HasPos Variable where

27.5.3 Showing

instance Show Variable where
27.5.4 Collecting variables
instance HasVariables Variable where

27.5.5 Grounding
instance Groundable a => Groundable [a] where

27.5.6 Kind inference
instance HasKind Variable where

27.5.7 DeepSeq
instance NFData Variable where

28 Logic.Arguments
Module ABR.Logic.Arguments implements arguments for logic systems.
module ABR.Logic.Arguments (Argument(..), argumentP) where

28.1 Data type
An Argument of an atom may be either:
- a constant (Const);
- a variable (Var); or
- a tuple containing constants and/or variables.

```haskell
data Argument =
  Const { arConst :: Constant, arPos :: Pos } | Var { arVar :: Variable, arPos :: Pos } |
  Tuple { arElems :: [Argument], arPos :: Pos }
```

The order of a tuple’s elements must be maintained. A tuple has at least 2 elements. A nested tuple is equivalent to a flat one.

\[(x, y, z) = (x, y, x) = (x, (y, z))\]

28.2 Parsers
```
argument ::= constant
  | variable
  | "(" argument {"," argument}+ ")".
```

```
argumentP :: Parser Argument
```

28.3 Instance declarations

28.3.1 Positions
instance HasPos Argument where

28.3.2 Comparing
instance Eq Argument where instance Ord Argument where

28.3.3 Showing
instance Show Argument where

28.3.4 Collecting constants
instance HasConstants Argument where

28.3.5 Collecting variables
instance HasVariables Argument where

28.3.6 Grounding
instance Groundable Argument where

28.3.7 Kind inference
instance HasKind Argument where

28.3.8 DeepSeq
instance NFData Argument where

29 Logic.Atoms
Module ABR.Logic.Atoms implements atoms.
module ABR.Logic.Atoms (Atom(..), atomNameP, atomP, HasAtoms(..)) where

29.1 Data type
An Atom is a proposition symbol, Prop. An atom may have a list of arguments.

```haskell
data Atom =
  Prop { aName :: String, aArgs :: [Argument], aPos :: Pos }
```

29.2 Parsers
```
atomName ::= lName | uName.
```

```
atomNameP :: Parser (String, Pos)
```

```
argList ::= "(" argument {"," argument} ")".
```

```
argListP :: Parser [Argument]
```

```
specialAtom ::= argument (< | <= | ==) argument.
```

September 12, 2018

atomP recognises atoms.

atomP :: Parser Atom

29.3 Collecting atoms

It is required for various purposes to identify all of the distinct atoms that occur in an object. Atoms can be collected from instances of class HasAtoms.

class HasAtoms a where

  getAtoms x A adds any atoms in x to A.

  getAtoms :: a -> S.Set Atom -> S.Set Atom

29.4 Instance declarations

29.4.1 Comparing

  instance Eq Atom where

  instance Ord Atom where

29.4.2 Positions

  instance HasPos Atom where

29.4.3 Showing

  instance Show Atom where

29.4.4 Collecting Atoms

  instance HasAtoms Atom where

29.4.5 Collecting constants

  instance HasConstants Atom where

29.4.6 Collecting variables

  instance HasVariables Atom where

29.4.7 Grounding

  instance Groundable Atom where

29.4.8 DeepSeq

  instance NFData Atom where

30 Logic.Literals

Module ABR.Logic.Literals implements literals.

module ABR.Logic.Literals
    (Literal(..), pLiteralP, Negatable(..), Complementable(..)
    ) where

30.1 Data type

A Literal is any atom a (Pos) or its negation ¬a (Neg).

data Literal =
  Pos { lAtom :: Atom,
         lPos :: Pos
   } |
   Neg { lAtom :: Atom,
         lPos :: Pos
   }

30.2 Parser

literal ::= atom | "~" literal | "(" literal ")".

pLiteralP recognises literals.

pLiteralP :: Parser Literal

30.3 Negation

Class Negatable includes types that may be logically negated with ¬.

class Negatable a where

  neg x negates x. For example if a is an atom, neg a = ¬a, and
  neg ¬a = a.

  neg :: a -> a

  pos x returns the “positive” x. For example if a is an atom, pos a =
  pos ¬a = a.

  pos :: a -> a

Class Complementable includes types that may be logically complemented with ¬.

class Complementable a where

  comp x complements x. For example if a is an atom, comp a =
  ¬a = ¬¬a.

  comp :: a -> a

30.4 Instance declarations

30.4.1 Comparing

  instance Eq Literal where

  instance Ord Literal where

30.4.2 Positions

  instance HasPos Literal where

30.4.3 Showing

  instance Show Literal where

30.4.4 Negation

  instance Negatable Literal where

  instance Complementable Literal where

30.4.5 Collecting Atoms

  instance HasAtoms Literal where

30.4.6 Collecting constants

  instance HasConstants Literal where
30.4.7 Collecting variables
instance HasVariables Literal where

30.4.8 Grounding
instance Groundable Literal where

30.4.9 DeepSeq
instance NFData Literal where

31 Logic.QuineMcClusky
Module ABR.Logic.QuineMcClusky implements the Quine-
McCluskey algorithm for simplifying boolean expressions as
described in Rosen[3].

module ABR.Logic.QuineMcClusky (  
  QMBit(..), qmSimplify
) where

31.1 Data types
A Quine-McCluskey bit (QMBit) is either zero (Zer), one (One)
or a placeholder 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

31.2 Simplification
qmSimplify bss simplifies bss.

qmSimplify :: [[QMBit]] -> IO [[QMBit]]

31.3 Instance declarations

31.3.1 DeepSeq
instance Generic QMBit where { }
instance Generic QMBit where { }
instance Generic QMBit where { }

32 Graphics.Geometry
Module ABR.Graphics.Geometry implements some basic geometric
calculations.

{-# LANGUAGE TypeSynonymInstances,  
FlexibleInstances #-)
module ABR.Graphics.Geometry (  
  Point, Box, Angle,  
  GeoNum(  
    netBox, shiftBoxes, leastRightShift,  
    placeAroundOval, iGeo, iPoint, iBox, insetBox
  )  
) where

32.1 Data types
A Point on the plane in Cartesian coordinates \((x, y)\). It is assumed
that the coordinate system in 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 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 a are represented in degrees. Absolute angles are measured
anticlockwise from the positive \(x\)-axis.

type Angle a = a
Line a 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 a are represented by the start and end points.

type LineSeg a = (Point a, Point a)

32.2 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 \(\Delta_x, \Delta_y\) returns the boxes displaced by \(\Delta_x\) and
  shiftBoxes :: [Box a] -> a -> a -> [Box a]
  leastRightShift as bs no longer overlap list of boxes as, as in
figure 1.

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

distAroundOval \(n \circ \phi\) returns a list of \(n\) points distributed around
the oval inscribed in box \(a\). The points will be equally distributed
by angle of separation wrt to the centre of the oval, clockwise, and
starting from angle \(\phi\).

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

iGeo \(x\) converts a GeoNum \(x\) to an Int.

type GeoNum :: a -> Int
iGeo :: Int

iPoint \(p\) converts the coordinates of point \(p\) to Ints.

type GeoNum :: Point a -> Point Int
iPoint :: Point a -> Point Int

iBox \(b\) converts the coordinates of box \(b\) to Ints.

type GeoNum :: Box a -> Box Int
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
\(((x_1, y_1), (x_2, y_2))\).

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
do this.

September 12, 2018
32.3 Instance declarations
instance GeoNum Float where
instance GeoNum Double where
instance GeoNum Int where
instance GeoNum Integer where
instance GeoNum Rational where

33 Graphics.EPS
Module ABR.Graphics.EPS provides support for the composition of Encapsulated PostScript (EPS).
module ABR.Graphics.EPS (EPS, PS, BPS, bpsToEps, EPSDrawable(..),
    joinBPS, setUpFonts,
    times10Width, times10ItalWidth, helvetica10Width, helvetica10ObliqueWidth, helvetica10BoldWidth, helvetica10BoldObliqueWidth, symbol10Width, FontTag(..), FontString(..), ftWidth, fsWidth,
    (++-++), MakeFontTags(..), FontBlock(..),
    wrapWithinWidth, psStr, newpath, moveto, lineto, closepath, stroke, show_, arc, gcave, gcrestore, translate, box ) where

33.1 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)

33.2 Finalizing to EPS
bpsToEps b finalizes a BPS figure by reversing it and constructing the EPS header comment including the bounding box.
bsToEps :: BPS -> EPS

33.3 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

33.4 Merging BPS components
joinBPS a b Δx Δy puts figure b over figure a displaced by Δx and Δy.
joinBPS :: BPS -> BPS -> Double -> Double -> BPS

33.5 Drawing text
33.5.1 Switching fonts efficiently
setUpFonts is PS code to find the fonts and define procedures for switching to them.
setUpFonts :: PS

33.5.2 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
times10ItalWidth c returns the width of a character c in Times-Italic 10 point, in 72 dpi pixels. (not exhaustive)
times10ItalWidth :: 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

33.5.3 Font tags
Type FontTag tags a string (ftStr) with either:
• Space – a space between symbols at which lines may be broken.
• Times10 – Times font, roman face, 10 point;
• Times10Ital – 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.

Each tag may also optionally be underlined (ftUnder).
data FontTag =
    Space { ftUnder :: Bool } | Times10 { ftUnder :: Bool, ftStr :: String } | Times10Ital { ftUnder :: Bool, ftStr :: String } | Helvetica10 { ftUnder :: Bool, ftStr :: String } | Helvetica10Oblique { ftUnder :: Bool, ftStr :: String } | Helvetica10Bold { ftUnder :: Bool, ftStr :: String } | Helvetica10BoldOblique { ftUnder :: Bool, ftStr :: String } | Symbol10 { ftUnder :: Bool, ftStr :: String }
33.5.4 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.
```

33.5.5 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.
```

33.6 Text encoding

```
psStr cs encodes a string for inclusion in PostScript as a literal.
psStr :: String -> String
```

33.6 Conveniences

Some PostScript operators: `newpath`, `moveto`, `lineto`, `closepath`, `stroke`, `show`, `arc`, `gsave`, `grestore`, `translate`.

```
neuPath, moveto, lineto, closepath, stroke, show, arc, gsave, grestore, translate :: String
```

34 File.Lock

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

```
{-# language ScopedTypeVariables #-
module ABR.File.Lock (lockFile, unlockFile, isLockedFile, areAnyLocked, lockFiles, unlockFiles, lockGuard, blockGuard) where
```

34.1 Basic lock operations

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

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

```
extend :: String -> String
```

```
extend filename = filename ++ "\,LOCK"
```

34.2 Multiple file operations

```
areAnyLocked fs returns True if 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 ()
```

34.3 Guards

```
lockGuard directory fs handler process checks whether any of the files 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 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.
```

35 File.Versions

The `ABR.File.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.

```
{-# language ScopedTypeVariables #-
module ABR.File.Versions (readLatest, writeNew, writeNew', writeNew'', purgeVersions, getNames, readFile', writeFile', removeV, createDirectory', removeVersions, latestDate) where
```

35.1 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)
```

35.2 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)
```

September 12, 2018 24
35.3 Write the next version

```hs
writeNew :: FilePath -> FilePath -> String -> Bool -> IO ()
writeNew' :: FilePath -> FilePath -> String -> FileMode -> Bool -> IO ()
writeNew'' :: FilePath -> FilePath -> String -> Bool -> IO ()
```

writeNew 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 and the same userID and groupID as dir. Iff binary no text encoding is used.

writeNew' 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. Iff binary no text encoding is used.

writeNew'' 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-rw-rw-` and the same userID and groupID as dir. Iff binary no text encoding is used.

35.4 Purge old versions

```hs
purgeVersions :: FilePath -> IO ()
```

purgeVersions deletes all old versions of the file with the given root name in directory dir.

35.5 Remove all versions

```hs
removeVersions :: FilePath -> IO ()
```

removeVersions deletes all versions of the file with the given root name in directory dir.

35.6 Get all versions

```hs
getNames :: FilePath -> FilePath -> IO [String]
```

getNames returns the list of filenames in directory dir that contain the given root file name and a version number.

35.7 Read and write file bottlenecks

```hs
readFile' :: FilePath -> IO (Maybe String)
writeFile' :: FilePath -> FilePath -> String -> FileMode -> Bool -> IO ()
```

readFile' 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.

writeFile' writes binary content to dir/file. The new file has the assigned access permissions of dir (masked by `-rw-rw-rw-`) and the same userID and groupID as dir. Iff binary no text encoding is used.

35.8 Creating and removing directories

```hs
removeR :: FilePath -> IO ()
createDirectory' :: FilePath -> FilePath -> IO ()
```

removeR removes a file or directory, path, first, recursively removing all its contents if it is a directory. If path does not exists, nothing is done.

createDirectory' 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.

36 CGI

```hs
{-# language ScopedTypeVariables #-}
module ABR.CGI (printMimeHeader, docType, printDocType, put, put', HTag, HAttributes, baseE_, isindexE_, linkE_, metaE_, nextidE_, inputE_, hrE_, brE_, imgE_, ifndefN_, hrN_, brN_, h1E, h2E, h3E, h4E, h5E, h6E, htmlN, headN, titleN, bodyN, addressN, blockquoteN, d1N, d2N, d3N, h1N, h2N, h3N, h4N, h5N, h6N, htmlT, htmlError, getQueryString, getPathInfo, getScriptName, getScriptDirectory, getContentLength, getFormData, getFormData', dumpFormData ) where
```

36.1 Mime header

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

```hs
mimeHeader :: String
```

printMimeHeader :: IO ()

36.2 Document type

```hs
docType :: String
```

docType is printed by `printDocType` which is printed by `printMimeHeader`.

36.3 Special character encoding

```hs
put cs
```

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

36.4 HTML elements (generic)

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

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

36.5 HTML elements (specific shortcuts)

```hs
{-# language ScopedTypeVariables #-}
```

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

```hs
baseE_, isindexE_, linkE_, metaE_, nextidE_, inputE_, hrE_, brE_, imgE_, ifndefN_, hrN_, brN_, HAttributes = []
```

fooE_ attributes prints an empty element its tag and attributes. isindexN_, hrN_, brN_ prints an empty element with its tag and no attributes.
36.6 Standards

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

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

36.7 CGI inputs

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

`getPathInfo` returns the extra path info after the name of the CGI tool.

`getScriptDirectory` returns the URL of the CGI tool.

```haskell
getScriptDirectory :: IO String
```

37 Daytime

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

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

37.1 Data types

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

```haskell
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.

```haskell
data Weekday = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday
```

37.2 Lexing

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

```haskell
daytimeL :: Lexer
```

37.3 Parsing

A weekday has this case-insensitive syntax:

```haskell
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]].
```

37.5 Weekday methods

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

```
tomorrow, yesterday :: Weekday -> Weekday
```

37.6 Daytime methods

```
inInterval start duration time returns True if 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
```

38 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
```

38.1 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)]
```

38.2 Lexing scripts

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

```
programL performs the lexical analysis of any Haskell source. Ap-
ply deliterate to literate sources before lexing.
```

```
programL :: Lexer
```

38.3 Handling the offside rule

```
offside tlps applies the off-side layout rule, inserting braces and 
semicolons. tlps 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
```

38.4 Diagnostics

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

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

38.5 Poor man’s parsing

This section contains functions for analysing the results of the lex-
ing 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 tlps 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.

\texttt{discardInners \ tlp} filters out all less-than-top-level declarations, crudely by eliminating all \{stuff; stuff; \ldots ; stuff\} sequences inside the top-level such sequence in \texttt{tlps}.

\texttt{discardInners :: TLPs \rightarrow TLPs}

\texttt{discard \ tlp} discards all lexemes from a \texttt{tlp} list, \texttt{tlps}, up to but not including the next semi-colon or opening or closing brace.

\texttt{discard :: TLPs \rightarrow TLPs}

\texttt{moduleName \ tlp} extracts the name of the module from a \texttt{tlp} list, \texttt{tlps}, if there is one, or returns \{\} if there is none.

\texttt{moduleName :: TLPs \rightarrow TLPs}

\texttt{declarations \ tlp} a \texttt{tlp} list, \texttt{tlps}, up into its top level declarations.

\texttt{declarations :: TLPs \rightarrow \{TLPs\}}

\texttt{declared \ tlp} takes a top-level declaration \texttt{tlp} and returns the type of declaration (as a new set of \texttt{Tag}s), the names of the declared objects (\texttt{Lexemes}) and the positions of the names of the objects (\texttt{Pos}s).

\texttt{declared :: TLPs \rightarrow TLPs}

\section{39 Playing Cards}

Module \texttt{ABR.PlayingCards} provides basic data types for card playing games and problems.

\begin{verbatim}
module ABR.PlayingCards ( Suit(..), Rank(..), suits, ranks, Card(..), Deck, deck52, deck54, Hand, shuffle )

39.1 Data types

Most cards have of one of these \texttt{Suit}s: \texttt{Clubs \{♣\}; Diamonds \{♦\}; Hearts \{♥\}; Spades \{♠\}.}

\texttt{data Suit = Clubs | Diamonds | Hearts | Spades deriving (Eq, Ord, Enum, Bounded)}

Most cards have of one of these \texttt{Rank}s: \texttt{Ace \{A\}; R2 \{2\}; R3 \{3\}; R4 \{4\}; R5 \{5\}; R6 \{6\}; R7 \{7\}; R8 \{8\}; R9 \{9\}; R10 \{10\}; Jack \{J\}; Queen \{Q\}; King \{K\}.}

\texttt{data Rank = Ace \{A\} | R2 \{2\} | R3 \{3\} | R4 \{4\} | R5 \{5\} | R6 \{6\} | R7 \{7\} | R8 \{8\} | R9 \{9\} | R10 \{10\} | Jack \{J\} | Queen \{Q\} | King \{K\} deriving (Eq, Ord, Enum, Bounded)}

\texttt{ranks} is the set of all ranks. \texttt{suits} is the set of all suits.

\texttt{ranks :: [Rank]} \texttt{suits :: [Suit]}

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

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

A \texttt{Deck} of cards, or a \texttt{Hand} of cards:

\texttt{type Deck = [Card]}

\texttt{type Hand = [Card]}

39.2 Creating decks

\texttt{deck52} returns all the cards in a standard 52-card deck. \texttt{deck54} returns all the cards in a standard 52-card deck plus 2 jokers.

\texttt{deck52, deck54 :: Deck}

\texttt{shuffle \ deck} returns all the cards in \texttt{deck} in a new random order.

\texttt{shuffle :: Deck \rightarrow IO Deck}

39.3 Instance declarations

\texttt{instance Show Card where}

\texttt{instance Show Suit where}

\texttt{instance Show Rank where}

\texttt{instance Show Hand where}

\texttt{instance Show Deck where}

\section{40 Poker}

Module \texttt{ABR.Poker} provides basic stuff like categorisation of hands, not tactics.

\begin{verbatim}
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

40.1 Presorting and grouping

\texttt{sortByRankSuit \ cs} sorts \texttt{cs} by rank and then suit.

\texttt{sortBySuitRank \ cs} sorts \texttt{cs} by suit and then rank.

\texttt{sortByRankSuit, sortBySuitRank :: [Card] \rightarrow [Card]}

\texttt{groupByRank \ cs} groups the cards in \texttt{cs} by common ranks. Each group will be sorted by suit. All the groups are sorted by the length of the group. \texttt{groupBySuit \ cs} groups the cards in \texttt{cs} by common suits. Each group will be sorted by rank. All the groups are sorted by the length of the group.

\texttt{groupByRank, groupBySuit :: [Card] \rightarrow [[Card]]}

\texttt{areSuccRanks \ r \ r'} returns \texttt{True} iff \texttt{r'} is the next highest rank after \texttt{r}.

\texttt{areSuccRanks :: Rank \rightarrow Rank \rightarrow Bool}

\texttt{allSuccRanks \ H} returns \texttt{True} iff all the cards in \texttt{H} have consecutive ranks.

\texttt{allSuccRanks :: Hand \rightarrow Bool}

40.2 Categorisation of hands

A \texttt{HandType} is one of (in order of increasing value):

1. \texttt{Garbage} – not any of the other kinds, worth only the ranks of its cards;
2. \texttt{Pair} – two cards have the same rank and all of the other cards have different ranks;
3. \texttt{TwoPair} – there are two pairs of different ranks and the other card is yet another rank.
4. \texttt{Tripple} – three cards have the same rank and the rest other different ranks;
5. \texttt{Straight} – the cards all have sequential ranks (an ace can precede a deuce or follow a king) and some cards have different suits;
6. \texttt{Flush} – all cards are the same suit, but not with sequential ranks;
7. \texttt{FullHouse} – a triple and a pair;
8. \texttt{Poker} – four of a kind;
9. \texttt{StraightFlush} – all cards have sequential ranks and the same suit.

\texttt{data HandType = Garbage | Pair | TwoPair | Triple | Straight | Flush | FullHouse | Poker | StraightFlush deriving (Eq, Ord, Show, Enum)}

\texttt{handType \ H} classifies \texttt{H}.

\texttt{handType :: Hand \rightarrow HandType}

September 12, 2018
isGarbage, isPair, isTwoPair, 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

40.3 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 :: Card -> Card -> Ordering

compareCards :: [Card] -> [Card] -> Ordering

compareGroups :: [[Card]] -> [[Card]] -> Ordering

compareHands :: Hand -> Hand -> Ordering

better :: Hand -> Hand -> Ordering

beats, ties :: Hand -> Hand -> Bool

better :: Hand -> Hand -> Hand

40.4 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

41 SendMail

Module ABR.SendMail lets a Haskell program send an email.

module ABR.SendMail (sendMail) where

41.1 Function

sendMail whoTo subject content sends an email to whoTo about subject containing content.

sendMail :: String -> String -> String -> IO ()

42 (Experimental) Data.Graph

Module ABR.Data.Graph implements a directed, unweighted graph as an ADT. This is UNDER CONSTRUCTION. This implementation closely follows that described by Rabhi and Lapalme [4] and Launchbury [5].

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

42.1 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 vertices V 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.

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

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

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 :: (Ix v) => Graph v -> v -> [v]
43.5 Cycles detection

isCyclic \( g \) returns True if graph \( g \) is cyclic. A depth-first search is used.

\[
\text{isCyclic : (Ix v, Enum v) -> Graph v -> Bool}
\]

43 (Experimental) MySQL C API Binding

Module \text{ABR.MySQLBinding} 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.

\begin{verbatim}
module ABR.MySQLBinding (
    My_bool, My_ulonglong, MYSQL, MYSQL_RES,
    MYSQL_ROW, MYSQL_ROW_OFFSET, MYSQL_FIELD,
    MYSQL_FIELD_OFFSET, Enum_mysql_option,
    MYSQL_AFFECTED_ROWS, MYSQL_ROW_OFFSET, MYSQL_FIELD, MYSQL_RES,
    MYSQL_FIELD_OFFSET, Enum_mysql_option,
    mysql fetched, mysql error, mysql_close,
    mysql_row_seek, mysql_row_tell, mysql_select_db,
    mysql_real_escape_string, mysql_real_query,
    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_list_processes,
    mysql_list_fields, mysql_list_processes, mysql_list_tables,
    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_thread_id, mysql_use_result
) where

43.1 API Data types

43.1.1 Basics

A \textbf{My_bool} is a C char.

\begin{verbatim}
type My_bool = CChar
\end{verbatim}

A \textbf{My_ulonglong} is supposedly a 64 bit, unsigned integer, but the way is used implies signed is a more useful choice.

\begin{verbatim}
type My_ulonglong = CLong
\end{verbatim}

43.1.2 Connections

A \textbf{MYSQL} is some opaque C object. A pointer to this structure is our handle on a connection.

\begin{verbatim}
type MYSQL = 
\end{verbatim}

43.1.3 Results

A \textbf{MYSQL_RES} is some opaque C object. A pointer to this structure is our handle on a result to a query.

\begin{verbatim}
type MYSQL_RES = 
\end{verbatim}

43.1.4 Rows

A \textbf{MYSQL_ROW} is an array of strings. The strings are not terminated with \texttt{\textbackslash 0}, as they could be binary data.

\begin{verbatim}
type MYSQL_ROW = Ptr CString
\end{verbatim}

\begin{verbatim}
type MYSQL_ROW_OFFSET = is a pointer to a MYSQL_ROWS.
\end{verbatim}

43.1.5 Fields

A \textbf{MYSQL_FIELD} is a C structure.

\begin{verbatim}
type MYSQL_FIELD = 
\end{verbatim}

A \textbf{MYSQL_FIELD_OFFSET} is an offset into a MySQL field list.

\begin{verbatim}
type MYSQL_FIELD_OFFSET = CUInt
\end{verbatim}

43.1.6 Options

A \textbf{Enum_mysql_option} is a C enumeration, used by function mysql options.

\begin{verbatim}
type Enum_mysql_option = CUInt
\end{verbatim}

43.2 API Functions

\begin{verbatim}
mysql_affect_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_affect_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_affect_rows was called prior to calling mysql_store_result.
\end{verbatim}

\begin{verbatim}
mysql_affect_rows :: Ptr MYSQL -> IO My_ulonglong
\end{verbatim}

\begin{verbatim}
mysql_change_user mysql user passwd db changes the user to user with password 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.
\end{verbatim}

\begin{verbatim}
mysql_change_user :: Ptr MYSQL -> CChar -> CChar -> CChar -> IO My_bool
\end{verbatim}

\begin{verbatim}
mysql_character_set_name mysql returns the default character set for the current connection.
\end{verbatim}

\begin{verbatim}
mysql_character_set_name :: Ptr MYSQL -> IO CChar
\end{verbatim}

\begin{verbatim}
mysql_close mysql closes and deallocates the connection mysql.
\end{verbatim}

\begin{verbatim}
mysql_close :: Ptr MYSQL -> IO ()
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
mysql_data_seek :: Ptr MYSQL_RES -> CUInt -> IO ()
\end{verbatim}

\begin{verbatim}
mysql_errno mysql returns the error code returned by the last MySQL API function. 0 indicates no error.
\end{verbatim}

\begin{verbatim}
mysql_errno :: Ptr MYSQL_RES -> IO CUInt
\end{verbatim}

\begin{verbatim}
mysql_error mysql returns the error message returned by the last MySQL API function. An empty string indicates no error.
\end{verbatim}

\begin{verbatim}
mysql_error :: Ptr MYSQL_RES -> IO CChar
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
mysql_fetch_field :: Ptr MYSQL_RES -> IO (Ptr MYSQL_FIELD)
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}

September 12, 2018 30
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. Re-
turns NULL when there are no more rows to retrieve.

mysql_fetch_row :: Ptr MYSQL_RES -> IO (MYSQL_ROW)

mysql_field_count 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 -> 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 -> 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 suit-
ble 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 re-
turned.

mysql_init :: Ptr MYSQL -> IO CString

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

mysql_list_fields :: Ptr MYSQL -> 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 ex-
pression specified by the wild parameter. wild may contain the
wild-card characters '_', '*', or may be a NULL pointer to match
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_fields :: Ptr MYSQL_RES -> IO CInt

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_real_connect 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), ’\x’, ’r’, ’\n’, ’’ , ’\’, 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 con-
tain binary data, because binary data may contain the ’\0’ char-
acter. 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_tell result offset sets the row cursor to an arbitrary
row in a query result set. This requires that the result set struc-
ture contains the entire result of the query, so mysql_row_tell 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 sim-
ply 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_tell.

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

mysql_row_seek result returns the current position of the row cur-
sor 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_row_seek :: Ptr MYSQL_RES -> MYSQL_ROW_OFFSET
-> 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 informa-
tion 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 CInt

mysql_store_result mysql reads and returns a pointer to the en-
tire result for the last query, or NULL is 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 con-
nection. 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 CUInt

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)

44 (Experimental) MySQL Haskell API

Module ABB.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 ABB.MYSQL( MySQL, myConnect, myClose, myQuery, myFetch )

44.1 Data types

44.1.1 Connections

A MySQL is our handle on a MySQL connection.

type MySQL =

44.2 Functions

44.2.1 Establishing a connection

myConnect host user password db returns CheckFail mysql if a con-
nexion could be established to the MySQL server running on host
("host" = "localhost"), as user ("user" = the current user), with pass-
word password ("password" = 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))

44.2.2 Closing a connection

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

myClose :: MySQL -> IO ()

44.2.3 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))

44.2.4 Fetching query results

myFetch mysql 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, cols, css), where rows
is the number of rows in the data set, cols is the list of lengths of each
field for each row, and css 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.

September 12, 2018 32
45 (Experimental)  
Database.Relational

Module Database.Relational is UNDER CONSTRUCTION.

module ABR.Database.Relational where

45.1 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  
| Integer Integer  
| Double Double  
| 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 α(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]

45.2 Relational algebra

45.2.1 Projection of a tuple on a relation schema

The projection π_X(r) of a relation r on a relation schema X, is defined by:
1. X ⊆ α(r) and α(π_X(r)) = X
2. π_X(r) = \{ t[X] : t \in r \}

projectTable (X, p) r returns π_X(r) using p to project all the tuples in r.

projectTable :: (TableSchema, Projector) -> Table -> Table
proj X r returns π_X(r). If the schema restrictions are not met then the program will terminate with an error.
proj :: TableSchema -> Table -> Table

45.2.2 Projection of a relation on a relation schema

The projection π_X(r) of a relation r on a relation schema X, is defined by:
1. X ⊆ α(r) and α(π_X(r)) = X
2. π_X(r) = \{ t[X] : t \in r \}

projectTable (X, p) r returns π_X(r) using p to project all the tuples in r.

projectTable :: (TableSchema, Projector) -> Table -> Table
proj X r returns π_X(r). If the schema restrictions are not met then the program will terminate with an error.
proj :: TableSchema -> Table -> Table

45.2.3 Natural join

The natural join r_1 \& r_2 of relations r_1 and r_2 is defined by:
1. α(r_1 \& r_2) = α(r_1) \cup α(r_2)
2. r_1 \& r_2 = \{ t \in (r_1 \times r_2)[α(r_1 \& r_2)] : t[α(r_1)] \in r_1 \land t[α(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.


type Joiner = Row -> Row -> Maybe Row

makeJoiner X X' returns X \& X', 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 U X' if the overlapping attributes are equal.

makeJoiner :: TableSchema -> TableSchema  
-> (TableSchema, Joiner)

joinTables (X, j) r_1 r_2 returns r_1 \& r_2, provided X = α(r_1 \& r_2). j is used to join individual tuples.

joinTables :: (TableSchema, Joiner) -> Table -> Table  
-> Table
r_1 \&|\&| r_2 returns r_1 \& r_2.

infixl 5 \&|\&|
(|\&|\&|) :: Table -> Table -> Table

45.2.4 Union

The union r_1 \cup r_2 of relations r_1 and r_2 is defined by:
1. α(r_1 \cup r_2) = α(r_1) \cup α(r_2)
2. r_1 \cup r_2 = \{ t \in r_1 \lor t \in r_2 \}

r_1 \`u` r_2 returns r_1 \cup r_2, provided α(r_1) = α(r_2). No check is performed to ensure this precondition.

u :: Table -> Table -> Table

45.2.5 Difference

The difference r_1 \& r_2 of relations r_1 and r_2 is defined by:
1. α(r_1 \& r_2) = α(r_2)
2. r_1 \& r_2 = \{ t \in r_1 \land t \notin r_2 \}

r_1 \&`d` r_2 returns r_1 \& r_2, provided α(r_1) = α(r_2). No check is performed to ensure this precondition.

diff :: Table -> Table -> Table

45.2.6 Selection

The selection σ_p(r) of relation r by predicate p is defined by:
1. α(σ_p(r)) = α(r)
2. σ_p(r) = \{ t \in r : p(t[r]) \}

select p r returns σ_p(r).
select :: (Row -> Bool) -> Table -> Table
45.2.7 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. \( \rho_{B|A}(r) = \{t : t' \in r, \forall t'[B] = t[A] \land \forall C \in (\alpha(r) - \{A\}) \cdot t[C] = t'[C]\} \)

renameTable \( B \ A \ r \) returns \( \rho_{B|A}(r) \).

renameTable \( \cdot \) Attribute \( \rightarrow \) Attribute \( \rightarrow \) Table \( \rightarrow \) Table
renameTableSchema \( B \ A \ X \) returns \( (X - \{A\}) \cup \{B\} \).

renameTableSchema \( \cdot \) Attribute \( \rightarrow \) Attribute \( \rightarrow \) TableSchema

45.3 Datum operations

addDatum \( \cdot \) Datum \( \rightarrow \) Datum \( \rightarrow \) Datum

46 (Deprecated) Control.List

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

module ABR.Control.List
{-# DEPRECATED "Use Control.Applicative.\"\" instead. "#-#"
where

46.1 Backwards map

\( \text{pm} \) \( \text{js} \ x \) returns the list of results obtained by applying all the functions in \( \text{js} \) to \( x \).

\( \text{pm} :: [a \rightarrow b] \rightarrow a \rightarrow [b] \)

47 (Deprecated) Data.BSTree

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

module ABR.Data.BSTree
{-# DEPRECATED "Use Data.Map instead." #-#
where

47.1 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 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.

47.2 BSTree operations

emptyBST is an empty BSTree.

emptyBST :: Ord k \rightarrow BSTree k v

nullBST \( t \) returns True iff \( t \) is empty.

nullBST :: Ord k \rightarrow BSTree k v \rightarrow Bool

depthBST \( t \) returns the depth of a \( t \).

depthBST :: Ord k \rightarrow BSTree k v \rightarrow 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 \( (\lambda x \rightarrow x) \) to merely replace.

updateBST :: Ord k \rightarrow (v \rightarrow v \rightarrow v) \rightarrow k \rightarrow v
\rightarrow BSTree k v \rightarrow BSTree k v

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

deleteBST :: Ord k \rightarrow BSTree k v \rightarrow BSTree k v

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

lookupBST :: Ord k \rightarrow BSTree k v \rightarrow Maybe v

memberBST \( k \) \( t \) returns True iff \( k \) occurs in \( t \).

memberBST :: Ord k \rightarrow BSTree k v \rightarrow 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 \rightarrow BSTree a b \rightarrow [a] \rightarrow (a \rightarrow c)
\rightarrow ([b] \rightarrow c) \rightarrow c

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

flattenBST :: Ord k \rightarrow BSTree k v \rightarrow [(k,v)]

domBST \( t \) returns the list of keys in \( t \) in ascending order of key.

domBST :: Ord k \rightarrow BSTree k v \rightarrow [k]

ranBST \( t \) returns the list of values in \( t \) in ascending order of key.

ranBST :: Ord k \rightarrow BSTree k v \rightarrow [v]

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

pairs2BST :: Ord k \rightarrow [(k,v)] \rightarrow BSTree k v

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

list2BST :: Ord k \rightarrow [k] \rightarrow v \rightarrow BSTree k v

countBST \( t \) returns the number of elements in \( t \).

countBST :: Ord k \rightarrow BSTree k v \rightarrow Int

leftBST \( t \) returns the left-most element of \( t \).

rightBST \( t \) returns the right-most element of \( t \).

leftBST :: BSTree k v \rightarrow BSTree k v

rightBST :: BSTree k v \rightarrow BSTree k v

48 (Deprecated) 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
{-# DEPRECATED "Use Data.HashTable instead." #-#
where

48.1 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 =
48.2 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)

48.3 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 (\( \times _{} \rightarrow x \)) to merely replace the old value.

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

48.4 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)

48.5 Dumping a hash table

dumpHT :: Set printFun printFun HT returns 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 ()

49 (Deprecated) Data.Queue

The Data module implements the Queue ADT.

module Data.Queue
  (# DEPRECATED "Use Data.Sequence instead." #)
  (Queue, emptyQ, isEmptyQ, attachQ, frontQ, detachQ, extractQ,)

49.1 Data type

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

type Queue a =

49.2 Operations

emptyQ :: Queue a

isEmptyQ :: Queue a -> Bool

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

frontQ :: Queue a -> a

detachQ :: Queue a -> a

diffQ :: Queue a -> (a, Queue a)

extractQ :: Queue a -> [a]

50 (Deprecated) 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
  (# DEPRECATED "Use Data.Set instead." #)
  (SparseSet, emptySS, nullSS, insertSS, mkSS, deleteSS, elemSS, notElemSS, flattenSS, list2SS, countSS, isSubSet, unionSS, sectSS, diffSS )

50.1 Data type

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

type SparseSet a =

50.2 Operations

emptySS :: Ord k -> SparseSet k

nullSS :: Ord k -> Bool

insertSS :: Ord k -> SparseSet k -> k -> SparseSet k

mkSS :: Ord k -> SparseSet k

deleteSS :: Ord k -> SparseSet k -> k -> SparseSet k

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

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

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

flattenSS :: Ord k -> SparseSet k -> Bool

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

countSS :: SparseSet k -> Int

unionSS :: Ord k -> SparseSet k -> SparseSet k -> SparseSet k

sectSS :: Ord k -> SparseSet k -> SparseSet k -> SparseSet k

diffSS :: Ord k -> SparseSet k -> SparseSet k
51 (Deprecated) 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.

```haskell
module ABR.Data.Set

{-# DEPRECATED "Use Data.Set instead." #-}

Set a =

51.1 Data type

data Set a =

51.2 Operations

infixl 7 .&, .*, .*

infixl 7 .*, .<, .<=

infix 5 .<-, .<, .-<

51.3 Instances

51.3.1 Ord

instance Ord a =>_set a -> Set a where

51.3.2 Showing

instance Show a => Show (Set a) where

51.3.3 DeepSeq

51.3.4 DeepSeq

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

52 (Deprecated) 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

{-# DEPRECATED "Use Control.DeepSeq instead." #-}

(DeepSeq(...), ($!!)) where

52.1 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

52.2 Infix operator

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

infixr 0 'deepSeq', $!!

($!!) :: (DeepSeq a) => (a -> b) -> a -> b

52.3 Instance Declarations

52.3.1 Simple instances

instance DeepSeq () where ()

instance DeepSeq Bool where ()

instance DeepSeq Char where ()

instance DeepSeq Integer where ()

instance DeepSeq Int where ()

instance DeepSeq Double where ()
52.3.2 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

52.3.3 List instance

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

52.3.4 Maybe instance

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

52.3.5 Either instance

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

53 (Deprecated) DeepSeq.BSTree

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

module ABR.DeepSeq.BStree
{-# DEPRECATED
where

53.1 Instance declaration

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

54 (Deprecated) Logic.Qualification

Module ABR.Logic.Qualification implements qualification for CDL.

module ABR.Logic.Qualification
{-# DEPRECATED
"Ill-conceived, I think." #-}

54.1 Qualifiable class

Class Qualifiable overloads methods for qualifying names.

class Qualifiable a where

  qualify :: String -> a -> a

54.2 Instance declarations

54.2.1 Qualification

instance Qualifiable String where
updateConfig, 15
updateHT, 35
utcToLocalTime, 6
utcToZonedTime, 6

valueP, 16
Var, 20
varE, 26
Variable, 19
variable, 19
variableP, 19
variables
collecting, 20
varN, 26
vertaL, 11

warnMsg, 11
weCat, 5
Wednesday, 26
Weekday, 26
weekdayP, 26
WEMessage, 5
weMsg, 5
wePon, 5
WEResult, 5
whitespaceL, 12
wordWrap, 17
wrapWithinWidth, 24
writeFile’, 25
writeNew, 25
writeNew’, 25
writeNew‘’, 25

yesterday, 27
Zer, 22