Abstract

This document lists and describes the libraries developed for and common to the various systems I have developed in Haskell\(^1\), 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 4 3.6 Filename globbing 4

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

5 Util.Open 5
  5.1 Opening a URL 5

6 Util.Pos 5
  6.1 Positions in a source 6 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

\(^1\)ABRHLibs – a personal library of Haskell modules Copyright (C) 2007-2013 Andrew Rock

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

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

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser.Checks</td>
<td>12</td>
</tr>
<tr>
<td><strong>17.1</strong> Easy lexer and parser sequencing</td>
<td>12</td>
</tr>
<tr>
<td><strong>18 Parser.Predicates</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>18.1</strong> sNumber predicates</td>
<td>12</td>
</tr>
<tr>
<td><strong>19 Text.Configs</strong></td>
<td>13</td>
</tr>
<tr>
<td><strong>19.1</strong> Data types</td>
<td>13</td>
</tr>
<tr>
<td><strong>19.2</strong> Lexer</td>
<td>13</td>
</tr>
<tr>
<td><strong>19.3</strong> Parser</td>
<td>13</td>
</tr>
<tr>
<td><strong>19.4</strong> Showing</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.5</strong> Reading</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.6</strong> Accessing</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.6.1</strong> configPaths</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.6.2</strong> Lookup functions</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.7 Templates</strong></td>
<td>14</td>
</tr>
<tr>
<td><strong>19.7.1</strong> Simple Template Markup Language</td>
<td>14</td>
</tr>
<tr>
<td><strong>19.7.2</strong> Populating templates</td>
<td>14</td>
</tr>
<tr>
<td><strong>20 Text.JSON</strong></td>
<td>14</td>
</tr>
<tr>
<td><strong>20.1</strong> Basic data types</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.2</strong> Construction</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.3</strong> Lexer</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.4</strong> Parser</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.5</strong> Interrogation</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.6</strong> Instances</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.6.1</strong> Showing</td>
<td>15</td>
</tr>
<tr>
<td><strong>20.6.2</strong> Interrogation</td>
<td>15</td>
</tr>
<tr>
<td><strong>21 Text.CSV</strong></td>
<td>15</td>
</tr>
<tr>
<td><strong>21.1</strong> Basic data types</td>
<td>15</td>
</tr>
<tr>
<td><strong>21.2</strong> Parser</td>
<td>15</td>
</tr>
<tr>
<td><strong>21.3</strong> Interrogation</td>
<td>15</td>
</tr>
<tr>
<td><strong>21.4</strong> Instances</td>
<td>15</td>
</tr>
<tr>
<td><strong>21.4.1</strong> Showing</td>
<td>15</td>
</tr>
<tr>
<td><strong>21.4.2</strong> Interrogation</td>
<td>16</td>
</tr>
<tr>
<td><strong>22 Text.Markup</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>22.1</strong> Making text safe for HTML</td>
<td>16</td>
</tr>
<tr>
<td><strong>22.2</strong> Making text safe for LaTeX</td>
<td>16</td>
</tr>
<tr>
<td><strong>22.3</strong> Converting LaTeX to HTML</td>
<td>16</td>
</tr>
<tr>
<td><strong>23 Text.String</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>23.1</strong> Word wrapping</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.2</strong> Justification</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.3</strong> Tables with justified columns</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.4</strong> Fields</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.5</strong> Whitespace</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.6</strong> Pattern matching and substitution</td>
<td>16</td>
</tr>
<tr>
<td><strong>23.7</strong> Names</td>
<td>17</td>
</tr>
<tr>
<td><strong>23.8</strong> Path catenation operators</td>
<td>17</td>
</tr>
<tr>
<td><strong>23.9</strong> Simple String Delimitation</td>
<td>17</td>
</tr>
<tr>
<td><strong>24 Text.Showing</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>24.1</strong> Adding Delimiters</td>
<td>17</td>
</tr>
<tr>
<td><strong>24.2</strong> Controlling Precision</td>
<td>17</td>
</tr>
<tr>
<td><strong>25 Logic.Kinds</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>25.1</strong> Data type</td>
<td>17</td>
</tr>
<tr>
<td><strong>25.2</strong> Unification</td>
<td>17</td>
</tr>
<tr>
<td><strong>25.3</strong> Kind inference</td>
<td>17</td>
</tr>
<tr>
<td><strong>25.4</strong> Instance declarations</td>
<td>18</td>
</tr>
<tr>
<td><strong>25.4.1</strong> Showing</td>
<td>18</td>
</tr>
<tr>
<td><strong>26 Logic.Constaints</strong></td>
<td>18</td>
</tr>
<tr>
<td><strong>26.1</strong> Data types</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.2</strong> Parsers</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.3</strong> Collecting constants</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4</strong> Instance declarations</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.1</strong> Comparing</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.2</strong> Positions</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.3</strong> Showing</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.4</strong> Collecting constants</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.5</strong> Kind inference</td>
<td>18</td>
</tr>
<tr>
<td><strong>26.4.6</strong> DeepSeq</td>
<td>18</td>
</tr>
<tr>
<td><strong>27 Logic.Variables</strong></td>
<td>18</td>
</tr>
<tr>
<td><strong>27.1</strong> Data type</td>
<td>18</td>
</tr>
<tr>
<td><strong>27.2</strong> Parsers</td>
<td>18</td>
</tr>
<tr>
<td><strong>27.3</strong> Collecting variables</td>
<td>18</td>
</tr>
<tr>
<td><strong>27.4</strong> Grounding</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5</strong> Instance declarations</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.1</strong> Comparing</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.2</strong> Positions</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.3</strong> Showing</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.4</strong> Collecting variables</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.5</strong> Grounding</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.6</strong> Kind inference</td>
<td>19</td>
</tr>
<tr>
<td><strong>27.5.7</strong> DeepSeq</td>
<td>19</td>
</tr>
<tr>
<td><strong>28 Logic.Arguments</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>28.1</strong> Data type</td>
<td>19</td>
</tr>
<tr>
<td><strong>28.2</strong> Parsers</td>
<td>19</td>
</tr>
<tr>
<td><strong>28.3</strong> Instance declarations</td>
<td>19</td>
</tr>
<tr>
<td><strong>29 Logic.Atoms</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>29.1</strong> Data type</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.2</strong> Parsers</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.3</strong> Collecting atoms</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4</strong> Instance declarations</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.1</strong> Comparing</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.2</strong> Positions</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.3</strong> Showing</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.4</strong> Collecting Atoms</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.5</strong> Collecting constants</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.6</strong> Collecting variables</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.7</strong> Grounding</td>
<td>20</td>
</tr>
<tr>
<td><strong>29.4.8</strong> DeepSeq</td>
<td>20</td>
</tr>
<tr>
<td><strong>30 Logic.Literals</strong></td>
<td>20</td>
</tr>
<tr>
<td><strong>30.1</strong> Data type</td>
<td>20</td>
</tr>
<tr>
<td><strong>30.2</strong> Parsers</td>
<td>20</td>
</tr>
<tr>
<td><strong>30.3</strong> Negation</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4</strong> Instance declarations</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.1</strong> Comparing</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.2</strong> Positions</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.3</strong> Showing</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.4</strong> Negation</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.5</strong> Collecting Atoms</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.6</strong> Collecting constants</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.7</strong> Collecting variables</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.8</strong> Grounding</td>
<td>21</td>
</tr>
<tr>
<td><strong>30.4.9</strong> DeepSeq</td>
<td>21</td>
</tr>
<tr>
<td><strong>31 Logic.QuineMcClusky</strong></td>
<td>21</td>
</tr>
<tr>
<td><strong>31.1</strong> Data types</td>
<td>21</td>
</tr>
<tr>
<td><strong>31.2</strong> Simplification</td>
<td>21</td>
</tr>
<tr>
<td><strong>31.3</strong> Instance declarations</td>
<td>21</td>
</tr>
<tr>
<td><strong>31.3.1</strong> DeepSeq</td>
<td>21</td>
</tr>
<tr>
<td><strong>32 Graphics.Geometry</strong></td>
<td>21</td>
</tr>
<tr>
<td><strong>32.1</strong> Data types</td>
<td>21</td>
</tr>
<tr>
<td><strong>32.2</strong> Geometric computations</td>
<td>22</td>
</tr>
<tr>
<td><strong>32.3</strong> Instance declarations</td>
<td>22</td>
</tr>
<tr>
<td><strong>33 Graphics.EPS</strong></td>
<td>22</td>
</tr>
<tr>
<td><strong>33.1</strong> Data types</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.2</strong> Finalizing to EPS</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.3</strong> Drawing in BPS</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.4</strong> Merging BPS components</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.5</strong> Drawing text</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.5.1</strong> Switching fonts efficiently</td>
<td>22</td>
</tr>
<tr>
<td><strong>33.5.2</strong> Font metrics</td>
<td>23</td>
</tr>
<tr>
<td><strong>33.5.3</strong> Font tags</td>
<td>23</td>
</tr>
<tr>
<td><strong>33.5.4</strong> Font strings</td>
<td>23</td>
</tr>
<tr>
<td><strong>33.5.5</strong> Font blocks</td>
<td>23</td>
</tr>
<tr>
<td><strong>33.5.6</strong> Font encoding</td>
<td>23</td>
</tr>
</tbody>
</table>
33.6 Conveniences ........................................... 23
33.7 Instance declarations .................................. 23
33.7.1 EPSDrawable ........................................ 23

34 File.Lock .................................................. 23
34.1 Basic lock operations .................................... 24
34.2 Multiple file operations ................................. 24
34.3 Guards .................................................... 24

35 File.Versions ............................................... 24
35.1 Read the latest version .................................. 24
35.2 Date of the latest version ............................... 24
35.3 Write the next version ................................... 24
35.4 Purge old versions ....................................... 24
35.5 Remove all versions ..................................... 24
35.6 Get all versions ......................................... 24
35.7 Read and write file bottlenecks ....................... 24
35.8 Creating and removing directories .................... 24

36 CGI ........................................................ 24
36.1 Mime header ............................................. 25
36.2 Document type .......................................... 25
36.3 Special character encoding ............................ 25
36.4 HTML elements (generic) ............................. 25
36.5 HTML elements (specific shortcuts) ................. 25
36.6 Standards ............................................... 25
36.7 CGI inputs ............................................... 25

37 Daytime .................................................... 25
37.1 Data types ............................................... 25
37.2 Lexing .................................................... 26
37.3 Parsing ................................................... 26
37.4 Instance declarations .................................. 26
37.4.1 Showing ............................................. 26
37.4.2 Arithmetic .......................................... 26
37.5 Weekday methods ....................................... 26
37.6 Daytime methods ....................................... 26

38 HaskellLexer ................................................ 27
38.1 Handling literate scripts ............................... 27
38.2 Lexing scripts .......................................... 27
38.3 Handling the offside rule .............................. 27
38.4 Diagnostics .............................................. 27
38.5 Poor man's parsing ..................................... 27

39 Playing Cards ............................................... 27
39.1 Data types ............................................... 27
39.2 Creating decks ......................................... 27
39.3 Instance declarations .................................. 27

40 Poker ........................................................ 27
40.1 Presorting and grouping ............................... 28
40.2 Categorisation of hands ............................... 28
40.3 Comparison of hands ................................... 28
40.4 Hands with more than 5 cards ....................... 28

41 SendMail .................................................... 28
41.1 Function .................................................. 28

42 (Experimental) Data.Graph ................................ 28
42.1 Graph abstract data type ............................... 28
42.2 Sparse graph type ....................................... 29
42.3 Graph Operations ....................................... 29
42.4 Reachability ............................................. 29
42.5 Cycles detection ........................................ 29

43 (Experimental) MySQL C API Binding .................... 29
43.1 API Data types ......................................... 29
43.1.1 Basics ............................................... 29
43.1.2 Connections ........................................ 29
43.1.3 Results .............................................. 29
43.1.4 Rows ................................................ 29
43.1.5 Fields ............................................... 29
43.1.6 Options ............................................. 29
43.2 API Functions .......................................... 29

44 (Experimental) MySQL Haskell API ....................... 31
44.1 Data types .............................................. 31
44.1.1 Connections ........................................ 31
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 ..................... 32
45.1 Definitions and data types ............................ 32
45.2 Relational algebra ..................................... 32
45.2.1 Projection of a tuple on a relation schema ....... 32
45.2.2 Projection of a relation on a relation schema .. 32
45.2.3 Natural join ........................................ 32
45.2.4 Union ............................................... 33
45.2.5 Difference ......................................... 33
45.2.6 Selection .......................................... 33
45.2.7 Renaming .......................................... 33
45.3 Datum operations ...................................... 33

46 (Deprecated) Control.List ................................. 33
46.1 Backwards map ........................................ 33

47 (Deprecated) Data.BSearchTree .......................... 33
47.1 BSTree type ............................................ 33
47.2 BSTree operations ..................................... 33

48 (Deprecated) Data.HashTables ........................... 34
48.1 Data types .............................................. 34
48.2 Creating a new hash table ............................ 34
48.3 Updating an existing hash table ...................... 34
48.4 Looking up in a hash table ............................ 34
48.5 Dumping a hash table ................................ 34

49 (Deprecated) Data.Queue .................................. 34
49.1 Data type ............................................... 34
49.2 Operations .............................................. 34

50 (Deprecated) Data.SparseSet ............................. 34
50.1 Data type ............................................... 34
50.2 Operations .............................................. 35

51 (Deprecated) Data.Set .................................... 35
51.1 Data type ............................................... 35
51.2 Operations .............................................. 35
51.3 Instances ............................................... 35
51.3.1 Ord .................................................. 35
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 ..................................... 36
52.3.3 List instance ........................................ 36
52.3.4 Maybe instance ..................................... 36
52.3.5 Either instance ...................................... 36

53 (Deprecated) DeepSeq.BSearchTree ....................... 36
53.1 Instance declaration ................................... 36

54 (Deprecated) Logic.Qualification ......................... 36
54.1 Qualifiable class ....................................... 36
54.2 Instance declarations .................................. 36
54.2.1 Qualification ....................................... 36

References ................................................... 36
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), Deimos (an implementation of Defeasible Logic), and Phobos (an implementation of Plausible Logic).

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

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

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

2 Installation

2.1 Dependencies

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

```
cabal install glob
```

2.2 Building

This library is packaged for distribution in the file \texttt{ABRHLibs.zip}, available from \url{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 \texttt{.lit} and the \texttt{.lhs} Haskell sources derived from them). To extract the \texttt{.tex} and \texttt{.lhs} files from the \texttt{.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 \texttt{ABRHLibs/src}.

```
$ cd ABRHLibs/src
```

To compile the Haskell libraries, \texttt{ghc} is required. To typeset the documentation, the tools \texttt{bibtex} and \texttt{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 \LaTeX and Haskell sources, use

```
$ make CLEAN
```

To rebuild the file \texttt{ABRHLibs.zip}, use

```
$ make distribute
```

June 15, 2017

3 Util.Args

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

```groovy
{-# LANGUAGE CPP #-}
module ABR.Util.Args

     ( OptSpec(..), OptVal(..), Options, findOpts,
       assertFlagPlus, assertFlagMinus, deleteFlag,
       insertParam, deleteParam, emptyOptions,
       lookupFlag, lookupParam, lookupQueue

# if __GLASGOW_HASKELL__ < 706
   -- legacy for hobbit
# else
   , glob
# endif
 ) where
```

3.1 Data types

An \texttt{OptSpec} is used to specify the types of option expected.

```haskell
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 \texttt{FlagS name}. Users set or unset with \texttt{+name} or \texttt{-name} respectively.
- a parameter with a value. Specify with \texttt{ParamS name}. Users provide values with \texttt{-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 \texttt{QueueS name}. Users provide values with \texttt{-name value1 -name value2 ...}.

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

```haskell
data OptVal =
    FlagPlus | FlagMinus |
    ParamValue String | ParamMissingValue |
    ParamQueue (S.Seq String)

deriving Show
```

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

```haskell
type Options = M.Map String OptVal
```

3.2 Empty options

\texttt{emptyOptions} is an empty \texttt{Options}.

```haskell
emptyOptions :: Options
```

3.3 Option detection

\texttt{findOpts optSpecs args} returns \texttt{(options, leftovers)}, where:

- \texttt{optSpecs} is a list of option specifications; \texttt{args} is a list of command line arguments; \texttt{options} is a mapping from the option names to the values found; and \texttt{leftovers} is a list of any unconsumed arguments, typically file names.

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

3.4 Adding and deleting options

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

```haskell
assertFlagPlus, assertFlagMinus, deleteFlag ::
    String -> Options -> Options
```
Params can be inserted or deleted, with `insertParam` and `deleteParam`, respectively.

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

### 3.5 Looking up options

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

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

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

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

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

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

### 3.6 Filename globbing

- `glob paths` returns the really existing file names that match the paths which may contain wildcards.

```
# if __GLASGOW_HASKELL__ < 705
-- 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.

```
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.

```
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.

```
class ErrorCategory cat where
  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.

```
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 wem` returns a failing message, `wem`.
- `returnWarn wem` returns a warning message, `wem`.

### 4.5 Instances

- **Functor**
  ```
  instance Functor (WEResult cat) where
  returnPass = const . returnPass
  returnFail = returnFail
  returnWarn = returnWarn
  ```

- **Applicative**
  ```
  instance Applicative (WEResult cat) where
  returnPass = const . returnPass
  returnFail = returnFail
  returnWarn = returnWarn
  ```

- **Monad**
  Making a `WEResult` a monad makes possible the silent accumulation messages as the computation proceeds, and stopping when it’s failed.

```
instance Monad (WEResult cat) where
```

## 5 Util.Open

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

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

### 5.1 Opening a URL

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

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

## 6 Util.Pos

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

```
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”.

```
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.

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

getPos returns the position of \( x \).

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

6.2.1 Container instances

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

6.3 Relative positions

\( p_1 \) precedes \( p_2 \) if \( p_1 \) comes earlier in than \( p_2 \).

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

6.4 Format strings

A TimeFormat is a time format string.

```haskell
type TimeFormat = String
```

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

```haskell
format produces

```

6.5 Formatting times

```haskell
formatTime formats a time using the default locale.

```

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

```

6.6 Formated current time

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

```

6.7 Formated file modification time

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

```

6.8 Overloading for old/new time systems

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

```

```haskell
class LegacyTimes t where
    getCurrentLegacyTime gets the current time.
    getCurrentLegacyTime :: IO t
    t1 \( \cdot \) diffSec \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down (floor) to whole seconds.
    diffSec :: t \( \mapsto \) t \( \mapsto \) Int
    t1 \( \cdot \) diffMin \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole minutes.
    diffMin :: t \( \mapsto \) t \( \mapsto \) Int
    t1 \( \cdot \) diffHour \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole hours.
    diffHour :: t \( \mapsto \) t \( \mapsto \) Int
    t1 \( \cdot \) diffDay \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole days.
    diffDay :: t \( \mapsto \) t \( \mapsto \) Int
```

```haskell
formatLegacyTime returns a formatted legacy time in local time showing the time zone.

```

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.

```haskell
module ABR.Util.Time (LegacyTime, UTCTime, ZonedTime, 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`.

```
type LegacyTime =
```

7.2 Time conversions

```
utcToZonedTime converts a UTC time to a ZonedTime.

```

```haskell
utcToZonedTime :: UTCTime -> ZonedTime
```

```
utcToLocalTime converts a UTC time to a LocalTime.

```

```haskell
utcToLocalTime :: UTCTime -> LocalTime
```

7.3 Getting the current time

```
gGetCurrentUTCTime gets the current time in UTC.
```

```
gGetCurrentUTCTime :: IO UTCTime
```

```
gGetCurrentZonedTime gets the current local time and time zone.
```

```
gGetCurrentZonedTime :: IO ZonedTime
```

```
gGetCurrentLocalTime gets the current local time. Local times can not display the time zone on formatting.
```

```
gGetCurrentLocalTime :: IO LocalTime
```

7.4 Format strings

A TimeFormat is a time format string.

```
type TimeFormat = String
```

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

```
format produces
```

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

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

7.5 Formatting times

```
formatTime formats a time using the default locale.
```

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

```
formatUTCTime :: UTCTime -> String
```

7.6 Formated current time

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

```
currentTime :: String
```

7.7 Formated file modification time

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

```
fileModTime :: FilePath -> String
```

7.8 Overloading for old/new time systems

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

```
class LegacyTimes t where
    getCurrentLegacyTime gets the current time.
    getCurrentLegacyTime :: IO t
```

```
t1 \( \cdot \) diffSec \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down (floor) to whole seconds.
    diffSec :: t \( \mapsto \) t \( \mapsto \) Int
```

```
t1 \( \cdot \) diffMin \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole minutes.
    diffMin :: t \( \mapsto \) t \( \mapsto \) Int
```

```
t1 \( \cdot \) diffHour \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole hours.
    diffHour :: t \( \mapsto \) t \( \mapsto \) Int
```

```
t1 \( \cdot \) diffDay \( \cdot \) t2 returns \( t_1 - t_2 \) rounded down to whole days.
    diffDay :: t \( \mapsto \) t \( \mapsto \) Int
```

```
formatLegacyTime returns a formatted legacy time in local time showing the time zone.
```

```
formatLegacyTime :: IO String
```

7.9 Instances

7.9.1 LegacyTimes

```
instance LegacyTimes UTCTime where
```

```
# if __GLASGOW_HASKELL__ < 705
instance LegacyTimes ST.ClockTime where
```

```
# 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.²

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

8.1 Data type

The result of a Check a

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

data CheckResult passType failType = CheckPass passType | CheckFail failType

A Check takes some object and returns a CheckResult.

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

8.2 Sequencing checks

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

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

8.3 Parallel checks

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

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

A &? Check c is applied to each of the elements of xs in parallel, returning only the last result if all checks pass or all of the error messages catenated if any checks fail.

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

8.9 Data.List

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

module ABR.Data.List (merge, msort, split, cartProd, interleave, separate, fragments, fragments', dropEach, permutations, permutations', combinations, subBag, bagElem, powSet, powSet_, powSet', powSet'_, properSublists, pPlus, meet, disjoint, allUnique, duplicates, subset, isSubset, findSubset, noSuperSets, isSubSequence, notSubSequence, chop, chops, subSuffix, diff, osect, union, sortByLength, trimN, trim2, replace1, replaceAll) where

10 Data.Map


module ABR.Control.Map (lookupGuard) where
import qualified Data.Map as M

² Thanks to Daniel Young for suggested extensions to this module.
10.3 Bag-like operations

\( \text{subBag} \) \( \{ \cdot \} \) returns True if every element of \( xs \) occurs at least as frequently in \( ys \) as it does in \( xs \).

\( \text{bagElem} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{Bool} \)

\( \text{bagElem} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{Bool} \)

10.4 Set-like operations

\( \text{allUnique} \) \( \{ a \} \rightarrow \text{Bool} \)

\( \text{duplicates} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{Bool} \)

\( \text{snub} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{snub} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{isSubSet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{isSubSet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{isProperSubSet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{isProperSubSet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{oecet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{oecet} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)

\( \text{odiff} \) \( \{ a \} \rightarrow \{ [a] \} \rightarrow \text{a} \)
11 Data.NameTable

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

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

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.

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.

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.

lookupName :: NameTable -> String -> Maybe Int

11.4 Creating a name array

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

makeNameArray :: NameTable -> IO NameArray

12 Data.Supply

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

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.

newSupply :: a -> IO (Supply a)

12.3 Extracting values from a supply

supplyNext supply returns the next value from the supply.

supplyNext :: Supply a -> IO a

peekNext supply returns the next value from the supply, but does not change the supply, so that the next value extracted will be the same.

peekNext :: Supply a -> IO a

13 Debug.Array

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

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.

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

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

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

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

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

14 Debug.IArray

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

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.

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

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

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

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

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

15 Parser

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

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

```haskell
data Could a = Fail Pos Msg | Error Pos Msg | OK a
```

15.3 Analysers
An `Analyser` is a higher-level abstraction of both lexers and 
parsers. An analyser is a function that tries to accept a list of in-
puts 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`.

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

15.4 Elementary analysers
This is the simplest analyser. `succeedA` `v` succeeds with a prede-
termined value `v` and does not consume any input.

```haskell
succeed :: b -> Analyser a b
succeed v = [(v,0)]
```

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

```haskell
epsilonA :: Analyser a ()
epsilonA = [((),0)]
```

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

```haskell
failA :: Msg -> Analyser a b
failA msg = [(msg,0)]
```

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

```haskell
errorA :: Msg -> Analyser a b
errorA msg = [(msg,0)]
```

`endA` succeeds if there is no input left and returns the trivial value
`()`.3

```haskell
endA :: Analyser a ()
endA = 
```

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.

```haskell
(<|>) :: Analyser a b -> Analyser a b -> Analyser a b
(<|>) a1 a2 = a1 <|> a2
```

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

```haskell
(&>) :: Analyser a b -> Analyser a c -> Analyser a (b,c)
```

15.6 Analyser result modifiers
These functions modify an analyser by modifying the type of value 
it returns.

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

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

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

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

15.7 More analyser combinators
This definition of `cons`, as an uncurried form of : is used below.

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

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

```haskell
some :: Analyser a b -> Analyser 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.
```

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.

```haskell
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`.

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

June 15, 2017
The input to a Lexer function is a list of characters and their positions. The output from a lexer is a list of lexemes with their tags and positions and the list of unconsumed characters and their positions. The output could also be an error or failure message. By convention, a function that is a lexer has a name that ends with a capital L.

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

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

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

### 15.9 Elementary lexers

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

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

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

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

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

### 15.10 Special combinators for lexers

```
l1 %> tag overrides the tag produced by lexer l with tag.
( %>) :: Lexer -> Tag -> Lexer
```

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

```
<&&> :: Lexer -> Lexer -> Lexer
```

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

```
<&&> :: Lexer -> Lexer -> Lexer
```

```
l1 %> tag modifies a lexer l by catenation of all its returned lexemes and returning the supplied tag. The position returned is the first. This permits the use of the combinators some, many and optional (above).
```

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

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

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

```
tagFilter tag l modifies lexer l by making it throw out lexemes with a specified tag.
tagFilter :: Tag -> Lexer -> Lexer
```

### 15.11 Frequently used lexers

```
tokenL token recognizes a particular token.
tokenL :: String -> Lexer
```

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

This is a common lexical structure for sources:

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

---

June 15, 2017

11
newline ::= "\n".

vertab ::= "\v".

formfeed ::= "\f".

return ::= "\r".

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

whitespaceL recognizes any amount of whitespace, returning it with tag " ".

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

whitespaceL :: Lexer

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

dropWhite :: Lexer -> Lexer

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

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

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

cardinal ::= ($digit$)*.

cardinalL :: Lexer

fixedL recognizes an unsigned fractional number with no exponent.

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

fixedL :: Lexer

floatL recognizes an unsigned floating point number.

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

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 a String

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

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

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, but this format is nicer to edit by hand.

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

### 19.1 Data types

A configuration, `Config`, is one of:

- `CFlag` a flag that is set by its presence;
- `CParam` a parameter with an associated value;
- `CSet` a parameter with an associated set of configurations; or
- `CList` a parameter with an associated list of configurations.

```
data Config = CFlag String  
  | CParam String String  
  | CSet String Configs  
  | CList String [Configs]  
  deriving (Eq, Ord)
```

A `Configs` is a list of configurations.

```
type Configs = [Config]
```

### 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.

```
comment ::=  
  "#" ($anything not \n$) ("\n" | $end of file$);  
  level="lexical".
```

Names in configuration files may contain letters, digits, plus and minus signs, underscores, periods, bangs and slashes. Note that a number can lex as a `name`, as could many file paths. Names are case sensitive.

```
name ::= {$letter$ | $digit$ | "+" | "-" | "/" | "." | "!" | "/"}*
```

Strings are delimited by double quotes and may extend across many lines. Use two double quotes for one, à la Pascal.

```
string ::= $""$ (newline) ($anything not \"$) "$";  
  level="lexical".
```

The other symbols used are:

- `=` to bind a name to a value (either a `name` or a `string`), configuration set or configuration list;
- `{` to start a configuration set;
- `}` to close a configuration set;
- `[` to start a configuration list;
- `]` to close a configuration list; and
- `,` to separate items in a configuration list.

```
cfgsL :: Lexer
```

### 19.3 Parser

A value is either a `name` or a `string`.

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

Strings are delimited by double quotes and may extend across many lines. Use two double quotes for one, à la Pascal.

```
string ::= $""$ (newline) ($anything not \"$) "$";  
  level="lexical".
```

The other symbols used are:

- `=` to bind a name to a value (either a `name` or a `string`), configuration set or configuration list;
- `{` to start a configuration set;
- `}` to close a configuration set;
- `[` to start a configuration list;
- `]` to close a configuration list; and
- `,` to separate items in a configuration list.

```
cfgsL :: Lexer
```

A `configs` is a sequence of whitespace separated `cfgs`, parsed by `cfgsP`.

```
configs ::= {cfg};  
  level="grammar".
```

A `configs` is a sequence of whitespace separated `cfgs`, parsed by `cfgsP`.

```
configs ::= {cfg};  
  level="grammar".
```

A `cfgSet` is a `configs` in braces.

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

A `cfgList` is a comma separated sequence of `configs` in brackets.

```
configList ::= comma-separated configs  
  level="grammar".
```
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;
  - _DEFINED_, if the Config is a flag;
  - the unStringed text of a parameter;
  - _SET_, if the Config is a set; or
  - _LIST_, if the Config is a list.
- In the following sequences, no extra whitespace is permitted.
- The sequence #ifdef#configPath#text#ifndef# outputs the text iff configPath leads to a Config of any kind that exists, other- wise outputs nothing. The text has any template markup in it processed as usual.
- The sequence #ifndef#configPath#text#ifdef# outputs the text iff configPath does not lead to a Config that exists, otherwise outputs nothing. This text has any template markup in it processed as usual.
- The sequence #with#configPath#text#end# outputs the text iff configPath leads to a set Config that exists. The text has any template markup in it processed using the set as the new root Configs. If the configPath does not lead to a Config that exists, other- wise outputs nothing. The text has any template markup in it processed as usual.
- The sequence #foreach#configPath#text#end# outputs the text for each element of the list iff configPath leads to a list Config that exists. The text has any template markup in it processed using each element as its root Configs. If the configPath does not lead to a Config that exists, the output is _NOT_A_SET_. 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 :: Configs -> String

19.8 Text.JSON

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

module _ABR.Text.JSON_ (JString, jValue, jmString, jmNumber, jmArray, jmObject, lexerL, valueP, JQuery, JResult, module _ABR.Text.JSON_ )
20.1 Basic data types

A distinction must be made between string data that appears in the JSON test (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 descaping is the reverse process.

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

```haskell
data JPropertyName = String
```

A `JValue` is a lump of JSON data.

```haskell
data JValue =
```

20.2 Construction

```haskell
jmString :: String -> JValue
jmNumber :: (Num a, Show a) => a -> JValue
jmTrue, jmFalse, jmNull :: JValue
jmArray :: [JValue] -> JValue
jmObject :: [(JPropertyName, JValue)] -> JValue
```

20.3 Lexer

```haskell
lexerL :: Lexer
```

20.4 Parser

```haskell
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 `i`th element.
- `JQElements` – 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`.

```haskell
infixl 9 :->
data JQuery =
  JQStr |
  JQInt |
  JQDbe |
  JQBool |
  JQProp JPropertyName |
  JQProps |
  JQElem Int |
  JQElements |
  JQIsNull |
  JQuery :-> JQuery
```

Class `JResult` overloads `jGet`.

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

20.6 Instances

20.6.1 Showing

The `Show` instances reconstruct valid JSON syntax.

```haskell
instance Show JValue where
instance Show JName
```

20.6.2 Interrogation

```haskell
instance JResult String where
instance JResult Int where
instance JResult Double where
instance JResult Bool where
instance JResult JValue where
instance JResult [JValue] where
instance JResult [(JPropertyName, JValue)] where
```

21 Text.CSV

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

```haskell
module ABR.Text.CSV where
```

21.1 Basic data types

A CSV file is one big string.

```haskell
type CSV = String
```

A Field is one cell.

```haskell
type Field = String
```

A Row is one list of Fields.

```haskell
type Row = [Field]
```

A Table is a just list of Rows.

```haskell
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', makeHTMLEnSafe, makeHTMLEnSafe',  
  makeLaTeXSafe, latex2html  
} where

22.1 Making text safe for HTML

encodeHTML c returns c's special character encoding if c ∈ \{<, >, &\}, otherwise c. encodeHTML' returns c's special character encoding, additionally encoding all control characters.

encodeHTML, encodeHTML' :: Char -> String

makeHTMLEnSafe cs makes cs safe for HTML. This is not less than makeHTMLEnSafe'.

makeHTMLEnSafe cs, makeHTMLEnSafe' :: String -> String

22.2 Making text safe for LaTeX

makeLaTeXSafe cs makes cs safe for inclusion in a LATEX document as plain text, by encoding some special characters.

makeLaTeXSafe :: String -> String

22.3 Converting LaTeX to HTML

latex2html cs converts LATEX string cs to HTML. This is not meant for whole documents. It has some basics for writing comments just like this one. This is used by mashdoc.

latex2html :: String -> String

23 Text.String

Module ABR.Text.String is a collection of functions that operate on strings.

module ABR.Text.String {  
  wordWrap, lJustify, rJustify, lJustify', rJustify',  
  justifyColumn, makeTable, spaceColumns, makeTableL,  
  makeTableMR, fields, unfields, trim, nameCmp, nameLT,  
  fixNewlines, fixNewlines', spaces, findClosest,  
  (++)', (++.++), catenateWith, substs, subst,  
  Data.List.chop
  } where

23.1 Word wrapping

wordWrap width cs wraps the words in cs to no wider than width, unless a word is wider than width, returning a list of lines.

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

23.2 Justification

lJustify w cs pads cs with extra spaces on the right to make the overall width not less than w. rJustify w cs pads cs with extra spaces on the left to make the overall width not less than w.

lJustify, rJustify :: Int -> String -> String

lJustify' p w cs pads cs with extra pad characters p on the right to make the overall width not less than w. rJustify' p w cs pads cs with extra pad characters p on the left to make the overall width not less than w.

lJustify', rJustify' :: Char -> Int -> String -> String

23.3 Tables with justified columns

justifyColumn j col justifies all of the strings in col using j to justify them all to the same width, which is the width of the widest string in col.

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

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

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

spaceColumns cs cols spaces out columns cols by inserting columns of replicated strings cs.

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

makeTableL e cols makes a table from cols using all left justification, with e used to pad columns and separate columns.

makeTableL :: Char -> [[String]] -> String

makeTableMR js rows makes a table from elements that are themselves multi-rowed. js is a list of justifiers for each column (as in makeTable). rows is a list of rows, where each row is a list of columns.

makeTableMR :: [Int -> String -> String] -> [[[String]]] -> String

23.4 Fields

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

fields d e cs breaks string cs into a list of strings at each delimiter character d, removing escape characters e where appropriate. If the escaping is not required use ABR.Data.List.chop instead.

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

unfields d e css converts css into one string, with each field separated by the delimiter character d, adding escape characters e as needed. If the escaping is not required use Data.List.intersperse instead.

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

23.5 Whitespace

trim cs strips any whitespace from both ends of cs.

trim :: String -> String

fixNewlines cs rectifies the ends of lines in cs. It does not ensure that the last character is a newline.

fixNewlines :: String -> String

fixNewlines' cs rectifies the ends of lines in cs. This version ensures that the last line is complete, i.e. that unless cs is empty, the last character returned will be a newline.

fixNewlines' :: String -> String

spaces' n returns n spaces.

spaces :: Int -> String

23.6 Pattern matching and substitution

findClosest pattern candidates returns the position in candidates of the string which, ignoring case is closest to pattern or -1 if candidates is empty.

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

substs prs cs performs substitutions on cs. prs is a list of pairs (p, r), where p is a case sensitive pattern to be replaced by r wherever it occurs.
substs :: [(String, String)] -> String -> String
subst :: p r cs performs substitutions on cs. p is a case sensitive pattern to be replaced by r wherever it occurs.
subst :: String -> String -> String

subHashNums rs cs performs substitutions on cs. rs is a list of replacements. r!!0 will replace the pattern #0#. r!!1 will replace the pattern #1# ...
subHashNums :: [String] -> String -> String

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

subst :: String -> String -> String
More such operators can be constructed with
infixl 6 ++/++, ++.++
at the join are removed.

rs cs performs substitutions on cs. rs is a list of replacements. r!!0 will replace the pattern #0#. r!!1 will replace the pattern #1# ...
subHashNums :: [String] -> String -> String

23.7 Names

nameCmp n1 n2 orders n1 and n2. Use this to sort names with sortBy when names are in family-name comma other-names format.

nameCmp :: String -> String -> Ordering

nameLT n1 n2 returns True if name n1 < n2. 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 /s or .s at the join are removed.

infixl 6 ++/++, ++.++

++/++, ++.++ :: String -> String -> String

More such operators can be constructed with
catenateWith c cs cs', which catenates cs and cs' with exactly one c at the join.

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

23.9 Simple String Delimitation

unString s rectifies string s, by removing the double quotes from each end (if present) and replacing pairs of double quotes with just one. If there are no double quotes in s, it is returned unchanged.

unString :: String -> String

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

enString :: String -> String

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.

module ABR.Text.Showing ( showWithSep, showWithTerm, FormattedDouble(..), showFD format x = show (makeFormattedDouble format x)

24.1 Adding Delimiters

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

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

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

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.

data FormattedDouble =
FD (Maybe Int) Double
ED (Maybe Int) Double
GD (Maybe Int) Double

This Show instance applies the formatting to the Double.

instance Show FormattedDouble where

shows Prec _ fd = case fd of
FD md x -> showFFloat md x
ED md x -> showEFloat md x
GD md x -> showGFloat md x

makeFormattedDouble format x makes a FormattedDouble from a Double x and a string that describes the format, format, of the form (“f” | “e” | “g”) {digit}, e.g. “f2”.

makeFormattedDouble :: String -> Double -> FormattedDouble

25 Logic.Kinds

Module Kinds implements kinds for arguments in logic atoms.

module ABR.Logic.Kinds ( KNamed(..), HasKind(..), kindCheckList, kUnify ) where

25.1 Data type

An 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).

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

25.2 Unification

kUnify k k' returns the unification of k and k' if there is one.
kUnify :: Kind -> Kind -> Maybe Kind

25.3 Kind inference

Class HasKind overload functions pertaining to kinds.
class (Show a, HasPos a) => HasKind a where

kindCheck x infers the kind of x.

kindCheck :: a -> IO Kind

kindCheckList :: [a] -> IO Kind

June 15, 2017
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 ?.

instance Show Kind where

26 Logic.Constants

Module Logic.Constants implements constants.

module ABR.Logic.Constants (
    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).

data Constant =
    CNamed {
        cName :: String,
        cPos :: Pos
    }
  |
    CIntegral {
        cInt :: Integer,
        cPos :: Pos
    }
  |
    CString {
        cStr :: String,
        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 instantiated 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 \( v \rightarrow c \) replaces a variable \( v \) with a constant \( c \). Substitutions may be composed. \( s_1 \circ s_2 \) first performs \( s_1 \) and then \( s_2 \). NullSub is the null substitution that does nothing.

```haskell
data Substitution = NullSub
  | Variable :->- Constant
  | Substitution :>-> Substitution
deriving (Eq, Ord, Show)
```

Anything groundable should be an instance of class `Groundable`.

class Groundable a where
  ground1 v cx
    returns \( x \) with all occurrences of variable \( v \) replaced by constant \( c \).
  ground :: Substitution -> a -> a
  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.

```haskell
module ABR.Logic.Arguments ( Argument(..), argumentP ) where
```

28.1 Data type

An `Argument` of an atom may be either:

- a constant (`Constant`);
- 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

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

`argumentP` recognises arguments.

```haskell
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.

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

29.2 Parsers

`atomName` ::= lName | uName.

atomName

lName

uName

`atomNameP` recognises atom names.

atomNameP :: Parser (String, Pos)

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

argList

argument

argument

`argListP` recognises argument lists.

argListP :: Parser [Argument]

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

specialAtom

argument

argument

`atom ::= atomName [argList] | string | specialAtom.`

atom

atomName

argList

string

specialAtom

`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 ")".

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 :: a -> a

negates x. For example if a is an atom, neg a = ¬a, and neg ¬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[5].

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

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

deriving (Eq, Show)

31.2 Simplification

qmSimplify bss simplifies bss.

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

31.3 Instance declarations

31.3.1 DeepSeq

instance Generic QMBit where 

instance GNData QMBit where 

instance NFData 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 is conventional, with the y-axis the right way up, unlike most screen graphics coordinate systems. The actual numeric type is not specified, and where possible functions will be written to accommodate both any of Float, Double, Int, or Integer or Rational. See class GeoNum, below.

type Point a = (a, a)

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

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

Angles are represented in degrees. Absolute angles are measured anticlockwise from the positive x-axis.

type Angle a = a

Lines 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)

LineSegments 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.

instance GeoNum Rational where
instance GeoNum Integer where
instance GeoNum Int where
instance GeoNum Float where
instance GeoNum Double where

32.3 Instance declarations

instance GeoNum Float where
instance GeoNum Double where
instance GeoNum Integer where
instance GeoNum Rational 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.

33.2 Finalizing to EPS

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

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.

33.4 Merging BPS components

joinBPS a b Δx Δy puts figure b over figure a displaced by Δx and Δy.

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

Figure 1: The least right shift to stop two sets of boxes overlapping.
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 }
  deriving Show
```

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.
fsWidth :: FontString -> Double
```

```
f ++++ f' catenates FontStrings f and f', with a space added at
the join. If both of the FontTags at the joining ends are underlined,
the joining space will also be underlined.
(++++) :: FontString -> FontString -> FontString
fsWords f groups a FontString into the Space-separated
FontStrings.
fsWords :: FontString -> [FontString]
```

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

```
class MakeFontTags a where
    makeFontTags x renders x as a FontString
    makeFontTags :: a -> [FontTag]
    makeFontTagsPrec :: Int -> a -> [FontTag]
    makeFontTagsPrec :: Int -> a -> [FontTag]
```

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 con-
tains to within maximum width w if possible, returning the wrapped
FontBlock.
wrapWithinWidth :: Double -> FontString -> FontBlock
```

33.5.6 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 :: String
```

Draw a `box`.
```
box :: Double -> Double -> Double -> Double -> EPS
```

33.7 Instance declarations

33.7.1 EPSDrawable

interface EPSDrawable FontTag where

interface EPSDrawable FontString where

interface EPSDrawable FontBlock where

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` iff 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` iff 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` iff at least one of the files named in `fs` is locked.

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

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

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

### 34.3 Guards

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

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

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

```haskell
{-# language ScopedTypeVariables #-}

### 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.

```haskell
module ABR.File.Versions where
    readLatest, writeNew, writeNew', writeNew'',
    purgeVersions, getNames, readFile', writeFile',
    removeR, createDirectory', removeVersions,
    removeE, getNamesE, dirE, rootE, fileE, versionE,
    isLockedE, lockE, unlockE, purgeE, removeE, createE,
    readFile', writeFile', createDirectory', purgeE, removeE

    readLatest, writeNew, writeNew', writeNew''',
    purgeVersions, getNames, readFile', writeFile',
    removeE, createDirectory', removeVersions,
    removeE, getNamesE, dirE, rootE, fileE, versionE,
    isLockedE, lockE, unlockE, purgeE, removeE

    {-# language ScopedTypeVariables #-}
```

### 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)`

### 35.3 Write the next version

`writeNew` dir root content binary writes content to a new version of the file with the given root file name in directory `dir`. The new file has the given access mode and the same userID and groupID as `dir`. If binary no text encoding is used.

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

### 35.4 Purge old versions

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

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

### 35.5 Remove all versions

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

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

### 35.6 Get all versions

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

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

### 35.7 Read and write file bottlenecks

`readFile'` path provides a safe way to read a file without raising an exception if the file does not exist. It returns Nothing if the file could not be read, Just contents otherwise.

`readFile' :: FilePath -> IO (Maybe String)`

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

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

### 35.8 Creating and removing directories

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

`removeR :: FilePath -> IO ()`

`createDirectory' dir newDir creates a directory, newDir inside parentDir. If newDir already exists (as a file or directory) it is removed (along with its contents) first. newDir is assigned the same userID, groupID and access permissions as parentDir.

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

### 36 CGI

{-# language ScopedTypeVariables #-}

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

```haskell
module ABR.CGI (...
```

June 15, 2017 24

```haskell
```
36.1 Mime header
First things first. A CGI tool should print the magic lines identifying the output as HTML. \(\text{mimeHeader} \) is the MIME header text, which is printed by \(\text{printMimeHeader} \).

\[
\text{mimeHeader} :: \text{String} \\
\text{printMimeHeader} :: \text{IO} ()
\]

36.2 Document type
Identify the kind of html being generated. \(\text{docType} \) is printed by \(\text{printDocType} \).

\[
\text{docType} :: \text{String} \\
\text{printDocType} :: \text{IO} ()
\]

36.3 Special character encoding
\(\text{put} \) prints \(cs\) with all special characters encoded. \(\text{put'} \) prints \(cs\) encodes all control characters.

\[
\text{put} :: \text{cs} \rightarrow \text{IO} () \\
\text{put'} :: \text{String} \rightarrow \text{IO} ()
\]

36.4 HTML elements (generic)
HTML elements have a name (a \(\text{HTag}\)).

\[
\text{type HTag} = \text{String}
\]

HTML elements can have a list of attributes (\(\text{HAttributes}\)) of the form \(\text{name} = \text{value}\).

\[
\text{type HAttributes} = [(\text{String}, \text{String})]
\]

36.5 HTML elements (specific shortcuts)
This is not an exhaustive list. Add more as needed.

\(\text{tagE} \) attributes prints an empty element its \(\text{tag}\) and \(\text{attributes}\).

\[
\text{baseE}, \text{isindexE}, \text{linkE}, \text{metaE}, \text{nextidE}, \text{inputE}, \text{hrE}, \text{brE}, \text{imgE} :: \text{HAttributes} \rightarrow \text{IO} ()
\]

\(\text{tagG} \) prints an empty element with its \(\text{tag}\) and no attributes.

\[
\text{isindexG}, \text{hrG}, \text{brG} :: \text{IO} ()
\]

\(\text{tagA} \) attributes \(\text{contents}\) prints a non-empty element with its \(\text{tag}\), \(\text{attributes}\), and \(\text{contents}\).

\[
\text{htmlE}, \text{headE}, \text{titleE}, \text{styleE}, \text{bodyE}, \text{addressE}, \text{blockquoteE}, \text{formE}, \text{selectE}, \text{optionE}, \text{dirE}, \text{dlE}, \text{ddE}, \text{ddE}, \text{dlE}, \text{dirE}, \text{menuE}, \text{dirE}, \text{endE}, \text{areaE}, \text{citeE}, \text{codeE}, \text{emE}, \text{kbdE}, \text{sampE}, \text{strongE}, \text{varE}, \text{be}, \text{iE}, \text{ttE}, \text{uE}, \text{tableE}, \text{captionE}, \text{teE}, \text{theE}, \text{deE}, \text{subE}, \text{supE}, \text{centerE}, \text{fontE}, \text{smallE}, \text{bigE}, \text{textareaE}, \text{h1E}, \text{h2E}, \text{h3E}, \text{h4E}, \text{h5E}, \text{h6E} :: \text{HAttributes} \rightarrow \text{IO} () \rightarrow \text{IO} ()
\]

36.6 Standards
The \(\text{html}\) element needs to have certain attributes to meet standards. \(\text{htmlT}\) applies the attributes to go with the transitional doctype above.

\[
\text{htmlT} :: \text{IO} () \rightarrow \text{IO} ()
\]

36.7 CGI inputs

- \(\text{getQUERY_STRING}\) returns the text after the \(?)\) in a URL.
- \(\text{getPATH_INFO}\) returns the extra path info after the name of the CGI tool.
- \(\text{getSCRIPT_NAME}\) returns the URL of the CGI tool.
- \(\text{getScriptDirectory}\) returns the URL of the directory the CGI binary is in.
- \(\text{getGetString}, \text{getPathInfo}, \text{getScriptName}, \text{getScriptDirectory} :: \text{IO} \text{String}\)
- \(\text{getCONTENT_LENGTH}\) returns the number of bytes of content arriving via standard input.

\[
\text{getCONTENT_LENGTH} :: \text{IO} \text{Int}
\]

\(\text{getFormData}\) reads standard input to obtain the post method inputs, decodes them and returns them in a binary search tree.

\[
\text{getFormData} :: \text{IO} (\text{M.Map String String})
\]

\(\text{getFormData'}\) reads standard input to obtain the post method inputs in the \text{enctype}"multipart/form-data"\) format, decodes them and returns them in a binary search tree.

\[
\text{getFormData'} :: \text{IO} (\text{M.Map String String})
\]

For debugging forms: \(\text{dumpFormData}\) tree outputs the form inputs in a nicely encoded XHTML fragment.

\[
\text{dumpFormData} :: \text{M.Map String String} \rightarrow \text{IO} ()
\]

37 Daytime
Module \(\text{ABR.Daytime}\) provides time of day and weekday manipulations.

\[
\text{module ABR.Daytime (}
\text{Daytime(\ldots), weekday(\ldots), dayAndTimeP, dayAndTimeP, showDT24, inInterval, tomorrow, yesterday}
\text{)}
\]

37.1 Data types
A \text{Daytime} consists of: hours, \(\text{dtHrs}\), minutes, \(\text{dtMins}\); and seconds, \(\text{dtSecs}\).

\[
\text{data Daytime} = \text{Daytime} (\text{dtHrs :: Int}, \text{dtMins :: Int}, \text{dtSecs :: Int})
\]

A \text{Weekday} is one of: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday

\[
\text{data Weekday} = \text{Weekday (\ldots)}
\]

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

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

`daytimeL :: Lexer`

37.3 Parsing

A weekday has this case-insensitive syntax:

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

weekday parses a Weekday.

`weekdayP :: Parser Weekday`

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

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

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]].
```

daytime parses a Daytime.

`daytimeP :: Parser Daytime`

A day and time has this syntax:

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

dayAndTime parses a day and a time.

`dayAndTimeP :: Parser (Weekday,Daytime)`

37.4 Instance declarations

37.4.1 Showing

```
instance Show Daytime where

showDT :: Daytime -> String
```

37.4.2 Arithmetic

```
instance Num Daytime where

(*) :: Daytime -> Daytime -> Daytime
```

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 ≤ 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.

```haskell
code

38.1 Handling literate scripts

deliterate eps removes all informal text from eps, 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 complete for ASCII sources, but not for unicode sources.

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

programL :: Lexer
```

38.3 Handling the offside rule

```haskell
offside tlp s applies the off-side layout rule, inserting braces and semicolons. `tlp` 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

```haskell
unlex tlp s undoes all of the above good work by unrolling all of the lexing of `tlp`. 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 lexing phases above without using a real (combinator) `Parser`. This method might turn out to be good enough to generate the sort of information required to create the Haskell dictionary which started me down this path. I have also used it for logical line counting. Call this function `before` the next one. `promoteMethods` promotes the definitions within the where clause of class declarations to the top level.

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

39 Playing Cards

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

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

39.1 Data types

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

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

Most cards have one of these `Rank`s: `Ace`; `R2 ... R10`; `Jack`; `Queen`; `King`.

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

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

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

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

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

39.2 Creating decks

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

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

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

```haskell
shuffle :: Deck -> IO Deck
```

39.3 Instance declarations

```haskell
instance Show Suit where
instance Show Rank where
instance Show Card where

instance Show Hand where
```

40 Poker

Module `ABR.Poker` provides basic stuff like categorisation of hands, not tactics.

```haskell
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

sortByRankSuit \( cs \) sorts \( cs \) by rank and then suit.

sortBySuitRank \( cs \) sorts \( cs \) by suit and then rank.

sortByRankSuit, sortBySuitRank :: [Card] -> [Card]

groupByRank \( cs \) groups the cards in \( cs \) by common ranks. Each group will be sorted by suit. All the groups are sorted by the length of the group.

groupBySuit \( cs \) groups the cards in \( cs \) by common suits. Each group will be sorted by rank. All the groups are sorted by the length of the group.

groupByRank, groupBySuit :: [Card] -> [[Card]]

areSuccRanks \( r \rightarrow r' \) returns True iff \( r' \) is the next highest rank after \( r \).

areSuccRanks :: Rank -> Rank -> Bool

ties :: Hand -> HandType

allSuccRanks \( H \) returns True iff all the cards in \( H \) have consecutive ranks.

allSuccRanks :: Hand -> Bool

40.2 Categorisation of hands

A HandType is one of (in order of increasing value):

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

data HandType = Garbage | Pair | TwoPair | Triple | Straight | Flush | FullHouse | Poker | StraightFlush

handType \( H \) classifies \( H \).

handType :: Hand -> HandType

isGarbage, isPair, isTwoPair, isTriplet, isStraight, isFlush, isFullHouse, isPoker and isStraightFlush each return True iff some hand is that kind of hand.

isGarbage, isPair, isTwoPair, isTriplet, 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

cmpares :: Ord Card

cmpares :: [Card] -> [Card] -> Ordering

cmpares :: [Card] -> [Card] -> Ordering

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

compareHands :: Hand -> Hand -> Ordering

H beats \( H' \) iff \( H \) is a better poker hand than \( H' \). H ties \( H' \) if they have the same value.

better :: Hand -> Hand -> Bool

better \( H \) returns the better of hands \( H \) and \( H' \).

better :: Hand -> Hand -> Bool

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

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

class Graph g where

mkGraph :: Ix v -> E -> g v

vertices \( G \) returns the list of vertices \( V \) in graph \( G \).

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

boundsG \( G \) returns the least and greatest of the vertices \( V \) in graph \( G \).

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

edges \( G \) returns the list of edges \( E \) in graph \( G \).

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

adjacent \( G \) v returns the list of vertices in graph \( G \) that can be reached from vertex \( v \) in one step.
42.2 Sparse graph type

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

```haskell
newtype SGraph vertex =
  deriving (Show)
instance Graph SGraph where

42.3 Graph Operations

mapG f v v' G builds a new graph G' formed by applying f to every edge in G, such that:
1. G' = (V, E')
2. V' = [v..v']
3. E' = {f e | e ∈ E}

mapG :: (Ix v, Ix v', Graph g, Graph g') =>
  (Edge v -> Edge v') -> v' -> v' -> g v -> g' v'
transposeG G reverses all the edges in G.

42.4 Reachability

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

```
adjacent :: Ix v => g v -> v -> [v]
isAdjacent G v v' returns True iff vertex v' in graph G can be reached from vertex v in one step.

42.5 Cycles detection

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

adjacent :: Ix v => g v -> v -> [v]
isAdjacent G v v' returns True iff vertex v' in graph G can be reached from vertex v in one step.

43 (Experimental) MySQL C API Binding

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

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

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

43.1 API Data types

43.1.1 Basics

A `My_bool` is a C char.

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

```haskell
type My_ulonglong = CLLong
```

43.1.2 Connections

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

```haskell
type MYSQL =
```

43.1.3 Results

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

```haskell
type MYSQL_RES =
```

43.1.4 Rows

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

```haskell
type MYSQL_ROW = Ptr CString
```
A `MYSQL_ROW_OFFSET` is a pointer to a MYSQL_ROWS.

```haskell
type MYSQL_ROW_OFFSET =
```

43.1.5 Fields

A `MYSQL_FIELD` is a C structure.

```haskell
type MYSQL_FIELD =
```
A `MYSQL_FIELD_OFFSET` is an offset into a MySQL field list.

```haskell
type MYSQL_FIELD_OFFSET =
```

43.1.6 Options

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

```haskell
type Enum_mysql_option =
```

43.2 API Functions

`mysql_affected_rows` returns the number of rows changed by the last UPDATE, deleted by the last DELETE or inserted by the last INSERT statement. May be called immediately after `mysql_query` for UPDATE, DELETE, or INSERT statements. For SELECT statements, `mysql_affected_rows` works like `mysql_num_rows`. Returns an integer greater than zero to indicate the number of rows affected or retrieved. Zero indicates that no records were retrieved. A negative value is returned only if the query was a SELECT statement whose result could not be fetched. The value of `mysql_num_rows` was called prior to calling `mysql_store_result`.

```haskell
mysql_affected_rows :: Ptr MYSQL -> IO My_ulonglong
```
mysql_close mysql closes and deallocates the connection mysql.

mysql_close :: Ptr MYSQL -> IO ()

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

mysql_data_seek :: Ptr MYSQL_RES -> CULLong -> IO ()

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

mysql_errno :: Ptr MYSQL -> IO CUInt

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

mysql_error :: Ptr MYSQL -> IO CUInt

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

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

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

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

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

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

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

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

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

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

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

mysql_field_count :: Ptr MYSQL -> IO CUInt

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

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

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

mysql_free_result :: Ptr MYSQL_RES -> IO ()

mysql_get_client_info returns a string that represents the client library version.

mysql_get_client_info :: IO CString

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

mysql_get_host_info :: Ptr MYSQL -> IO CString

mysql_get_proto_info mysql returns the protocol version used by current connection.

mysql_get_proto_info :: Ptr MYSQL -> IO CUInt

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

mysql_get_server_info :: Ptr MYSQL -> IO CUInt

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

mysql_init :: Ptr MYSQL -> IO CPtr MYSQL

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

mysql_insert_id :: Ptr MYSQL -> IO CUInt mysql

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

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

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

mysql_list_dbs :: Ptr MYSQL -> CPtr MYSQL_RES

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

mysql_list_fields :: Ptr MYSQL -> CPtr 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 -> CPtr MYSQL_RES

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

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

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

44 (Experimental) MySQL

44.1 Data types

44.1.1 Connections
44.2 Functions

44.2.1 Establishing a connection

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

```haskell
myConnect :: String -> String -> String -> String
```

44.2.2 Closing a connection

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

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

44.2.3 Issuing a query

```haskell
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.
```

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

44.2.4 Fetching query results

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

```haskell
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.

45 (Experimental) Database.Relational

Module ABR.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.

```haskell
type Name = String
```

```haskell
data Attribute = AString Name
  | AInteger Name
  | ADouble Name
```

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.

```haskell
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.

```haskell
data Datum = DNull |
  DString String |
  DInteger Integer |
  DDouble Double
deriving (Eq, Ord, Show)
```

Within each row, the order of the data is significant.

```haskell
type Row = [Datum]
```

Each table has a schema that identifies the contents of each attribute.

```haskell
data Table = Table TableSchema [Row]
```

45.2 Relational algebra

45.2.1 Projection of a tuple on a relation schema

The projection π(t) of a tuple t on a relation schema X is computed by discarding the attributes in t that do not appear in X. This projection operation will be applied to many rows and is worth computing just once. A projector is a function that performs a projection on one tuple.

```haskell
type Projector = Row -> Row
```

A projector can be computed from the new and old table schemas.

```haskell
makeProjector Y X returns either:
  • Nothing, if the new schema Y contains an attribute not in the old schema X; or
  • Just Y′, p′, where table schema Y′ is the same as the requested new table schema Y with the exception that the attributes are in the same order as in the old table schema X, and p′ is the projector that can be applied to tuple that conforms to X to produce a tuple the conforms to Y′.
```

```haskell
makeProjector :: TableSchema -> TableSchema
  -> Maybe (TableSchema, Projector)
```

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. \( X \subseteq \alpha(r) \) and \( \alpha(\pi_X(r)) = X \)
2. \( \pi_X(r) = \{ t[X] : t \in r \} \)

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

```haskell
projectTable :: (TableSchema, Projector) -> Table -> Table
```

proj X r returns \( \pi_X(r) \). If the schema restrictions are not met then the program will terminate with an error.

```haskell
proj :: TableSchema -> Table -> Table
```

45.2.3 Natural join

The natural join \( r_1 \bowtie r_2 \) of relations r_1 and r_2 is defined by:

1. \( \alpha(\bowtie_{r_1 \bowtie r_2} = \alpha(\langle r_1 \rangle) \cup \alpha(\langle r_2 \rangle) \)
2. \( r_1 \bowtie r_2 = \{ t \in (r_1 \times r_2) : [\alpha(t)] \subseteq r_1 \land [\alpha(t)] \subseteq r_2 \} \)

Each tuple of the joined relation contains the combined attributes from two tuples, one from each of the original relations. A joined tuple is only formed if the overlapping attributes were equal.

A joiner is a function that joins two tuples if the overlapping attributes are equal.

```haskell
type Joiner = Row -> Row -> Maybe Row
```

June 15, 2017 32
The \( \sigma \) function in relation \( r \) is defined by:
1. \( \alpha(\sigma_p(r)) = \alpha(r) \)
2. \( \sigma_p(r) = \{ t \in r : p(t) \} \)

select \( \sigma_p(r) \) returns \( \sigma_p(r) \).

select :: (Row -> Bool) -> Table -> Table

46.1 Backwards map

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

\( \text{pam} :: [a -> b] -> a -> [b] \)

47 (Deprecated) Data.BSTree

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

module ABR.Data.BSTree

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

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

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 \texttt{key} must be an instance of type class \texttt{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 => BSTree k v
nullBST t returns True iff \( t \) is empty.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ranBST :: Ord k => BSTree k v -> [v]
48  (Deprecated) Data.HashTables
Module ABR.Data.HashTables implements hash tables in an efficient 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." #-}
(HashTable, newHT, updateHT, lookupHT, dumpHT)

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 (\( \times \mapsto x \)) 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)

49  (Deprecated) Data.Queue
The ABR.Data.Queue module implements the Queue ADT.

module ABR.Data.Queue
{-# DEPRECATED "Use Data.Sequence instead." #-}
(Queue, emptyQ, isNothingQ, attachQ, frontQ, detachQ, extractQ)

49.1 Data type
A Queue is a first-in-first-out sequence.
type Queue a =

49.2 Operations
emptyQ is an empty queue.
emptyQ :: Queue a

isNothingQ q returns True iff queue q is empty.
isNothingQ :: Queue a -> Bool

attachQ e q attaches e to the back of queue q.
attachQ :: a -> Queue a -> Queue a

frontQ q returns the value at the front of queue q.
frontQ :: Queue a -> a

detachQ q returns the element that was at the front of queue q and the q after that element has been detached.
detachQ :: Queue a -> (a, Queue a)

extractQ q returns the list of all elements in queue q.
extractQ :: Queue a -> [a]

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 implemented with a height-balanced tree.
type SparseSet a =
50.2 Operations

emptySS :: Ord k => SparseSet k -> Bool
nullSS :: Ord k => SparseSet k -> Bool
insertSS :: Ord k => k -> SparseSet 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
isSubSet1 :: Ord k => SparseSet k -> SparseSet k -> Bool
flattenSS :: Ord k => SparseSet k -> [k]
list2SS :: Ord k => [k] -> SparseSet k
nullSS :: Ord k => SparseSet k -> Bool
emptySS :: Ord k => SparseSet k

50.2.1 Ord

Ord instances for sets are very similar to those for lists.

emptySS = nullSS = True
nullSS (insertSS x ss) = (notElemSS x ss)
insertSS x (deleteSS y ss) = (deleteSS y ss) . (elemSS x)

50.3 Instances

Ord i => SparseSet i

51 (Deprecated) Data.Set

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

module ABR.Data.Set
{-# DEPRECATED "Use Data.Set instead." #-}
(:set, set, set, set, setHidden, str, ((.),), (\&), (\.), (+), (<), (<=), card, sany, snull, sall, sall, sall, sall, sall)

51.1 Data type

data Set a =

51.2 Operations

infix7 .\k, \*, .*
infix6 .\|, \-, .\*
infix 5 .<, .<, .<

eset is {}.
eset :: Set a
take xs returns the set of elements in xs.
set :: (Ord a) => [a] -> Set a
seti x returns [x].
setii a -> Set a
unseti {x} returns x.
unseti :: Set a -> a
list A returns the list of elements in A.
list :: Set a -> [a]
A .\| B returns A \cup B. A .\& B returns A \cap B. A .\< B returns A \subset B.
A .\< B returns True if A \subset B. A .\< B returns True if A \subset B.
A \< B returns True if A \subset B. A \< B returns True if A \subset B.
A \< B returns True if A \subset B. A \< B returns True if A \subset B.
A \< B returns True if A \subset B. A \< B returns True if A \subset B.
card A returns |A|.
card :: Set a -> Int
smap f A returns \{fx : x \in A\}.

51.3 Instances

51.3.1 Ord

instance (Ord a) => Set a
51.3.2 Showing

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

51.3.3 DeepSeq

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

52 (Deprecated) DeepSeq

Module `ABR.DeepSeq` was pinch 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

Class `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 `deepSeq` x deeply evaluates `x` and then returns `f x`.

`deepSeq` 0 'deepSeq', `deepSeq`

`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 Ordering where
instance DeepSeq Integer where
instance DeepSeq Int where
instance DeepSeq Float 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.


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." #-}

(Qualifiable..)

where

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

References
