

KiCS2

The Kiel Curry System (Version 2)

User Manual

Version 0.5.0 of 2015-11-30

Michael Hanus¹ [editor]

Additional Contributors:

Bernd Braßel²

Björn Peemöller³

Fabian Reck⁴

Jan Rasmus Tikovsky⁵

(1) University of Kiel, Germany, mh@informatik.uni-kiel.de

(2) University of Kiel, Germany, bbr@informatik.uni-kiel.de

(3) University of Kiel, Germany, bjp@informatik.uni-kiel.de

(4) University of Kiel, Germany, fre@informatik.uni-kiel.de

(4) University of Kiel, Germany, jrt@informatik.uni-kiel.de

Contents

Preface	5
1 Overview of KiCS2	6
1.1 Installation	6
1.2 General Use	6
1.3 Restrictions	7
1.4 Modules in KiCS2	8
2 Using the Interactive Environment of KiCS2	9
2.1 Invoking KiCS2	9
2.2 Commands of KiCS2	9
2.3 Options of KiCS2	11
2.4 Source-File Options	15
2.5 Using KiCS2 in Batch Mode	16
2.6 Command Line Editing	16
2.7 Customization	16
2.8 Emacs Interface	16
3 Extensions	17
3.1 Narrowing on Int Literals	17
3.2 Recursive Variable Bindings	17
3.3 Functional Patterns	18
3.4 Order of Pattern Matching	19
4 Recognized Syntax of Curry	20
4.1 Notational Conventions	20
4.2 Lexicon	20
4.2.1 Case Mode	20
4.2.2 Identifiers and Keywords	20
4.2.3 Comments	21
4.2.4 Numeric and Character Literals	21
4.3 Layout	22
4.4 Context Free Grammar	23
5 Optimization of Curry Programs	27
6 CurryDoc: A Documentation Generator for Curry Programs	28
7 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs	31
8 CurryTest: A Tool for Testing Curry Programs	33

9	CurryPP: A Preprocessor for Curry Programs	35
9.1	Integrated Code	35
9.1.1	Regular Expressions	36
9.1.2	Format Specifications	36
9.1.3	HTML Code	37
9.1.4	XML Expressions	38
9.2	Sequential Rules	38
9.3	Default Rules	39
10	runcurry: Running Curry Programs	40
11	ERD2Curry: A Tool to Generate Programs from ER Specifications	42
12	Spicey: An ER-based Web Framework	43
13	Technical Problems	44
	Bibliography	45
A	Libraries of the KiCS2 Distribution	47
A.1	AbstractCurry and FlatCurry: Meta-Programming in Curry	47
A.2	General Libraries	48
A.2.1	Library AllSolutions	48
A.2.2	Library Assertion	49
A.2.3	Library Char	50
A.2.4	Library Combinatorial	52
A.2.5	Library Constraint	52
A.2.6	Library CPNS	53
A.2.7	Library CSV	54
A.2.8	Library Debug	54
A.2.9	Library Directory	55
A.2.10	Library Distribution	56
A.2.11	Library Either	61
A.2.12	Library ErrorState	61
A.2.13	Library FileGoodies	63
A.2.14	Library FilePath	64
A.2.15	Library Findall	67
A.2.16	Library Float	68
A.2.17	Library Function	70
A.2.18	Library GetOpt	71
A.2.19	Library Global	73
A.2.20	Library GUI	74
A.2.21	Library Integer	86
A.2.22	Library IO	88
A.2.23	Library IOExts	90

A.2.24	Library JavaScript	92
A.2.25	Library KeyDatabaseSQLite	95
A.2.26	Library List	100
A.2.27	Library Maybe	104
A.2.28	Library NamedSocket	105
A.2.29	Library Parser	106
A.2.30	Library Pretty	107
A.2.31	Library Profile	120
A.2.32	Library Prolog	122
A.2.33	Library PropertyFile	123
A.2.34	Library Read	124
A.2.35	Library ReadNumeric	124
A.2.36	Library ReadShowTerm	125
A.2.37	Library SearchTree	126
A.2.38	Library SetFunctions	129
A.2.39	Library Socket	133
A.2.40	Library System	134
A.2.41	Library Time	135
A.2.42	Library Unsafe	138
A.2.43	Library UnsafeSearchTree	138
A.2.44	Library ValueSequence	140
A.3	Data Structures and Algorithms	141
A.3.1	Library Array	141
A.3.2	Library Dequeue	142
A.3.3	Library FiniteMap	144
A.3.4	Library GraphInductive	147
A.3.5	Library Random	153
A.3.6	Library RedBlackTree	154
A.3.7	Library SCC	155
A.3.8	Library SetRBT	155
A.3.9	Library Sort	156
A.3.10	Library TableRBT	157
A.3.11	Library Traversal	158
A.3.12	Library Rewriting.Term	160
A.3.13	Library Rewriting.Substitution	161
A.3.14	Library Rewriting.Unification	162
A.3.15	Library Rewriting.UnificationSpec	162
A.4	Libraries for Web Applications	162
A.4.1	Library CategorizedHtmlList	162
A.4.2	Library HTML	163
A.4.3	Library HtmlCgi	176
A.4.4	Library HtmlParser	178
A.4.5	Library Mail	178
A.4.6	Library Markdown	179

A.4.7	Library URL	181
A.4.8	Library WUI	182
A.4.9	Library WUIjs	188
A.4.10	Library XML	197
A.4.11	Library XmlConv	199
A.5	Libraries for Meta-Programming	206
A.5.1	Library AbstractCurry.Types	206
A.5.2	Library AbstractCurry.Files	211
A.5.3	Library AbstractCurry.Select	213
A.5.4	Library AbstractCurry.Build	216
A.5.5	Library AbstractCurry.Pretty	219
A.5.6	Library FlatCurry.Types	222
A.5.7	Library FlatCurry.Files	228
A.5.8	Library FlatCurry.Goodies	229
A.5.9	Library FlatCurry.Pretty	241
A.5.10	Library FlatCurry.Read	246
A.5.11	Library FlatCurry.Show	246
A.5.12	Library FlatCurry.XML	247
A.5.13	Library FlatCurry.FlexRigid	247
A.5.14	Library FlatCurry.Compact	248
A.5.15	Library FlatCurry.Annotated.Types	250
A.5.16	Library FlatCurry.Annotated.Pretty	251
A.5.17	Library FlatCurry.Annotated.Goodies	254
A.5.18	Library FlatCurry.Annotated.TypeSubst	267
A.5.19	Library FlatCurry.Annotated.TypeInference	268
A.5.20	Library CurryStringClassifier	270
B	Markdown Syntax	272
B.1	Paragraphs and Basic Formatting	272
B.2	Lists and Block Formatting	273
B.3	Headers	275
C	Auxiliary Files	276
D	External Operations	278
	Index	281

Preface

This document describes KiCS2 (**K**iel **C**urry **S**ystem **V**ersion **2**), an implementation of the multi-paradigm language Curry [11, 20] that is based on compiling Curry programs into Haskell programs. Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). The current KiCS2 implementation does not support concurrent constraints. Alternatively, one can write distributed applications by the use of sockets that can be registered and accessed with symbolic names. Moreover, KiCS2 also supports the high-level implementation of graphical user interfaces and web services (as described in more detail in [12, 13, 14, 17]).

We assume familiarity with the ideas and features of Curry as described in the Curry language definition [20]. Therefore, this document only explains the use of the different components of KiCS2 and the differences and restrictions of KiCS2 (see Section 1.3) compared with the language Curry (Version 0.9.0). The basic ideas of the implementation of KiCS2 can be found in [10, 9].

Acknowledgements

This work has been supported in part by the DFG grants Ha 2457/5-1 and Ha 2457/5-2.

1 Overview of KiCS2

1.1 Installation

This version of KiCS2 has been developed and tested on Linux systems. In principle, it should be also executable on other platforms on which a Haskell implementation (Glasgow Haskell Compiler and Cabal) exists, like in many Linux distributions, Sun Solaris, or Mac OS X systems.

Installation instructions for KiCS2 can be found in the file `INSTALL.txt` stored in the KiCS2 installation directory. Note that there are two possibilities to install KiCS2:

Global installation: KiCS2 is installed in some global system directory where users have no write permission. In this case, some options for experimenting with KiCS2 (like `supply` or `ghc`, see below) are not available (since they require the recompilation of parts of the installed system).

Local installation: KiCS2 is installed in some local user directory where the user has write permission and the option `GLOBALINSTALL` in the `Makefile` of the KiCS2 installation is set as follows:

```
GLOBALINSTALL=no
```

In this case, all options of KiCS2 are available.

Furthermore, KiCS2 can be installed with experimental support for profiling of executables. To use profiling, two requirements have to be met:

- The libraries that are shipped with the GHC that is used by KiCS2 have to be installed with profiling enabled. This is the default for the system libraries contained in the GHC release, but may not be the case for additional libraries.
- The `Makefile` of KiCS2 contains an option `PROFILING` which has to be set to `yes` to enable profiling support. You may either change the `Makefile` to

```
PROFILING = yes
```

or specify this setting while starting the installation process using

```
make <optional target> PROFILING=yes
```

In the following, *kics2home* denotes the installation directory of the KiCS2 installation.

1.2 General Use

All executables required to use the different components of KiCS2 are stored in the directory *kics2home*/bin. You should add this directory to your path (e.g., by the `bash` command “`export PATH=kics2home/bin:$PATH`”).

The source code of the Curry program must be stored in a file with the suffix “.curry”, e.g., `prog.curry`. Literate programs must be stored in files with the extension “.lcurry”.

Since the translation of Curry programs with KiCS2 creates some auxiliary files (see Section C for details), you need write permission in the directory where you have stored your Curry programs. The auxiliary files for all Curry programs in the current directory can be deleted by the command

```
cleancurry
```

(this is a shell script stored in the `bin` directory of the KiCS2 installation, see above). The command

```
cleancurry -r
```

also deletes the auxiliary files in all subdirectories.

1.3 Restrictions

There are a few minor restrictions on Curry programs when they are processed with KiCS2:

- *Singleton pattern variables*, i.e., variables that occur only once in a rule, should be denoted as an anonymous variable “_”, otherwise the parser will print a warning since this is a typical source of programming errors.
- KiCS2 translates all *local declarations* into global functions with additional arguments (“lambda lifting”, see Appendix D of the Curry language report). Thus, in the various run-time systems, the definition of functions with local declarations look different from their original definition (in order to see the result of this transformation, you can use the Curry-Browser, see Section 7).
- Tabulator stops instead of blank spaces in source files are interpreted as stops at columns 9, 17, 25, 33, and so on. In general, tabulator stops should be avoided in source programs.
- Encapsulated search: The general definition of encapsulated search of the Curry report [19] is not supported. Thus, the corresponding prelude operations like `try`, `solveAll`, `once`, `findAll`, or `best` are not defined in the KiCS2 prelude. However, KiCS2 supports appropriate alternatives to encapsulate non-deterministic computations:

Strong encapsulation: This means that all potential non-determinism is encapsulated. Since this might result in dependencies on the evaluation strategy (see [7] for a detailed discussion), this kind of encapsulation is only available as I/O operations. For instance, the library `AllSolutions` (Section A.2.1) defines the operation

```
getAllValues :: a → IO [a]
```

to compute all values of a given argument expression. There is also the library `SearchTree` (Section A.2.37) which supports user-programmable search strategies and contains some predefined strategies like depth-first, breadth-first, iterative deepening search.

Weak encapsulation: This means that only the non-determinism defined inside an encapsulation operator is encapsulated. Conceptually, these operators are offered as *set functions* [2] which compute the set of all results but do not encapsulate non-determinism in the actual arguments. See the library `SetFunctions` (Section A.2.38) for more details.

- Concurrent computations based on the suspension of expressions containing free variables are not yet supported. KiCS2 supports *value generators* for free variables so that a free variable is instantiated when its value is demanded. For instance, the initial expression

```
x == True where x free
```


is non-deterministically evaluated to `False` and `True` by instantiating `x` to `False` and `True`, respectively. Thus, a computation is never suspended due to free variables. This behavior also applies to free variables of primitive types like integers. For instance, the initial expression

```
x*y:=1 where x,y free
```

is non-deterministically evaluated to the two solutions

```
{x = -1, y = -1} True
{x = 1, y = 1} True
```

- Unification is performed without an occur check.
- There is currently no general connection to external constraint solvers.

1.4 Modules in KiCS2

KiCS2 searches for imported modules in various directories. By default, imported modules are searched in the directory of the main program and the system module directory “*kics2home/lib*”. This search path can be extended by setting the environment variable `CURRYPATH` (which can be also set in a KiCS2 session by the option “`:set path`”, see below) to a list of directory names separated by colons (“:”). In addition, a local standard search path can be defined in the “.kics2rc” file (see Section 2.7). Thus, modules to be loaded are searched in the following directories (in this order, i.e., the first occurrence of a module file in this search path is imported):

1. Current working directory (“.”) or directory prefix of the main module (e.g., directory “/home/joe/curryprogs” if one loads the Curry program “/home/joe/curryprogs/main”).
2. The directories enumerated in the environment variable `CURRYPATH`.
3. The directories enumerated in the “.kics2rc” variable “`libraries`”.
4. The directory “*kics2home/lib*”.

The same strategy also applies to modules with a hierarchical module name with the only difference that the hierarchy prefix of a module name corresponds to a directory prefix of the module. For instance, if the main module is stored in directory `MAINDIR` and imports the module `Test.Func`, then the module stored in `MAINDIR/Test/Func.curry` is imported (without setting any additional import path) according to the module search strategy described above.

Note that the standard prelude (*kics2home/lib/Prelude.curry*) will be always implicitly imported to all modules if a module does not contain an explicit import declaration for the module `Prelude`.

2 Using the Interactive Environment of KiCS2

This section describes the interactive environment KiCS2 that supports the development of applications written in Curry. The implementation of KiCS2 contains also a separate compiler which is automatically invoked by the interactive environment.

2.1 Invoking KiCS2

To start KiCS2, execute the command “`kics2`” (this is a shell script stored in `kics2home/bin` where `kics2home` is the installation directory of KiCS2). When the system is ready (i.e., when the prompt “`Prelude>`” occurs), the prelude (`kics2home/lib/Prelude.curry`) is already loaded, i.e., all definitions in the prelude are accessible. Now you can type various commands (see next section) or an expression to be evaluated.

One can also invoke KiCS2 with parameters. These parameters are usual a sequence of commands (see next section) that are executed before the user interaction starts. For instance, the invocation

```
kics2 :load Mod :add List
```

starts KiCS2, loads the main module `Mod`, and adds the additional module `List`. The invocation

```
kics2 :load Mod :eval config
```

starts KiCS2, loads the main module `Mod`, and evaluates the operation `config` before the user interaction starts. As a final example, the invocation

```
kics2 :load Mod :save :quit
```

starts KiCS2, loads the main module `Mod`, creates an executable, and terminates KiCS2. This invocation could be useful in “make” files for systems implemented in Curry.

There are also some specific options that can be used when invoking KiCS2:

`--noreadline` (if used, this must be the first option): Do not use input line editing (see Section 2.6).

`-Dname=val` (these options must come before any KiCS2 command): Overwrite values defined in the configuration file “`.kics2rc`” (see Section 2.7), where `name` is a property defined in the configuration file and `val` its new value.

2.2 Commands of KiCS2

The **most important commands** of KiCS2 are (it is sufficient to type a unique prefix of a command if it is unique, e.g., one can type “`r`” instead of “`:reload`”):

`:help` Show a list of all available commands.

`:load prog` Compile and load the program stored in `prog.curry` together with all its imported modules.

`:reload` Recompile all currently loaded modules.

`:add $m_1 \dots m_n$` Add modules m_1, \dots, m_n to the set of currently loaded modules so that their exported entities are available in the top-level environment.

`:expr` Evaluate the expression $expr$ to normal form and show the computed results. In the default mode, all results of non-deterministic computations are printed. One can also print first one result and the next result only if the user requests it. This behavior can be set by the option `interactive` (see below).

Free variables in initial expressions must be declared as in Curry programs. In order to see the results of their bindings,¹ they must be introduced by a “`where...free`” declaration. For instance, one can write

```
not b where b free
```

in order to obtain the following bindings and results:

```
{b = False} True
{b = True} False
```

Without these declarations, an error is reported in order to avoid the unintended introduction of free variables in initial expressions by typos.

If the free variables in the initial goal are of a polymorphic type, as in the expression

```
xs++ys:= [z] where xs,ys,z free
```

they are specialized to the type “`()`” (since the current implementation of KiCS2 does not support computations with polymorphic logic variables).

`:eval $expr$` Same as `:expr`. This command might be useful when putting commands as arguments when invoking `kics2`.

`:quit` Exit the system.

There are also a number of **further commands** that are often useful:

`:type $expr$` Show the type of the expression $expr$.

`:programs` Show the list of all Curry programs that are available in the load path.

`:cd dir` Change the current working directory to dir .

`:edit` Load the source code of the current main module into a text editor. If the variable `editcommand` is set in the configuration file “`.kics2rc`” (see Section 2.7), its value is used as an editor command, otherwise the environment variable “`EDITOR`” is used as the editor program.

`:edit m` Load the source text of module m (which must be accessible via the current load path if no path specification is given) into a text editor which is defined as in the command “`:edit`”.

¹Currently, bindings are only printed if the initial expression is not an I/O action (i.e., not of type “`IO...`”) and there are not more than ten free variables in the initial expression.

`:show` Show the source text of the currently loaded Curry program. If the variable `showcommand` is set in the configuration file “.kics2rc” (see Section 2.7), its value is used as a command to show the source text, otherwise the command “cat” is used.

`:show m` Show the source text of module *m* which must be accessible via the current load path if no path specification is given.

`:source f` Show the source code of function *f* (which must be visible in the currently loaded module) in a separate window.

`:source m.f` Show the source code of function *f* defined in module *m* in a separate window.

`:browse` Start the CurryBrowser to analyze the currently loaded module together with all its imported modules (see Section 7 for more details).

`:interface` Show the interface of the currently loaded module, i.e., show the names of all imported modules, the fixity declarations of all exported operators, the exported datatypes declarations and the types of all exported functions.

`:interface m` Similar to “:interface” but shows the interface of the module *m* which must be in the load path of KiCS2.

`:usedimports` Show all calls to imported functions in the currently loaded module. This might be useful to see which import declarations are really necessary.

`:set option` Set or turn on/off a specific option of the KiCS2 environment (see 2.3 for a description of all options). Options are turned on by the prefix “+” and off by the prefix “-”. Options that can only be set (e.g., `path`) must not contain a prefix.

`:set` Show a help text on the possible options together with the current values of all options.

`:save` Save the currently loaded program as an executable evaluating the main expression “main”. The executable is stored in the file `Mod` if `Mod` is the name of the currently loaded main module.

`:save expr` Similar as “:save” but the expression *expr* (typically: a call to the main function) will be evaluated by the executable.

`:fork expr` The expression *expr*, which is typically of type “IO ()”, is evaluated in an independent process which runs in parallel to the current KiCS2 process. All output and error messages from this new process are suppressed. This command is useful to test distributed Curry programs where one can start a new server process by this command. The new process will be terminated when the evaluation of the expression *expr* is finished.

`!:cmd` Shell escape: execute *cmd* in a Unix shell.

2.3 Options of KiCS2

The following options (which can be set by the command “:set”) are currently supported:

`path path` Set the additional search path for loading modules to *path*. Note that this search path is only used for loading modules inside this invocation of KiCS2.

The path is a list of directories separated by “:”. The prefix “~” is replaced by the home directory as in the following example:

```
:set path aux:~/tests
```

Relative directory names are replaced by absolute ones so that the path is independent of later changes of the current working directory.

`bfs` Set the search mode to evaluate non-deterministic expressions to breadth-first search. This is the default search strategy. Usually, all non-deterministic values are enumerated and printed with a breadth-first strategy, but one can also print only the first value or all values by interactively requesting them (see below for these options).

`dfs` Similarly to `bfs` but use a depth-first search strategy to compute and print the values of the initial expression.

`ids` Similarly to `bfs` but use an iterative-deepening strategy to compute and print the values of the initial expression. The initial depth bound is 100 and the depth-bound is doubled after each iteration.

`ids n` Similarly to `ids` but use an initial depth bound of *n*.

`parallel` Similarly to `bfs` but use a parallel search strategy to compute and print the values of the initial expression. The system chooses an appropriate number of threads according to the current number of available processors.

`parallel n` Similarly to `parallel` but use *n* parallel threads.

`prdfs` Set the search mode to evaluate non-deterministic expressions to primitive depth-first search. This is usually the fastest method to print *all* non-deterministic values. However, it does not support the evaluation of values by interactively requesting them.

`choices n` Show the internal choice structure (according to the implementation described in [10]) resulting from the complete evaluation of the main expression in a tree-like structure. This mode is only useful for debugging or understanding the implementation of non-deterministic evaluations used in KiCS2. If the optional argument *n* is provided, the tree is shown up to depth *n*.

`supply i` (not available in global installations, see Section 1.1) Use implementation *i* as the identifier supply for choice structures (see [10] for a detailed explanation). Currently, the following values for *i* are supported:

integer: Use unbounded integers as choice identifiers. This implementation is described in [10].

ghc: Use a more sophisticated implementation of choice identifiers (based on the ideas described in [6]) provided by the Glasgow Haskell Compiler.

pureio: Use IO references (i.e., memory cells) for choice identifiers. This is the most efficient implementation for top-level depth-first search but cannot be used for more sophisticated search methods like encapsulated search.

ioref (default): Use a mixture of `ghc` and `pureio`. IO references are used for top-level depth-first search and `ghc` identifiers are used for encapsulated search methods.

`vn` Set the verbosity level to n . The following values are allowed for n :

$n = 0$: Do not show any messages (except for errors).

$n = 1$: Show only messages of the front-end, like loading of modules.

$n = 2$: Show also messages of the back end, like compilation messages from the Haskell compiler.

$n = 3$: Show also intermediate messages and commands of the compilation process.

$n = 4$: Show also all intermediate results of the compilation process.

`prompt p` Sets the user prompt which is shown when KiCS2 is waiting for input. If the parameter p starts with a letter or a percent sign, the prompt is printed as the given parameter, where the sequence “`%s`” is expanded to the list of currently loaded modules and “`%`” is expanded to a percent sign. If the prompt starts with a double quote, it is read as a string and, therefore, also supports the normal escape sequences that can occur in Curry programs. The default setting is

```
:set prompt "%s> "
```

`+/-interactive` Turn on/off the interactive mode. In the interactive mode, the next non-deterministic value is only computed when the user requests it. Thus, one has also the possibility to terminate the enumeration of all values after having seen some values.

`+/-first` Turn on/off the first-only mode. In the first-only mode, only the first value of the main expression is printed (instead of all values).

`+/-optimize` Turn on/off the optimization of the target program.

`+/-bindings` Turn on/off the binding mode. If the binding mode is on (default), then the bindings of the free variables of the initial expression are printed together with the result of the expression.

`+/-time` Turn on/off the time mode. If the time mode is on, the cpu time and the elapsed time of the computation is always printed together with the result of an evaluation.

`+/-trace` Turn on/off the trace mode. If the trace mode is on, it is possible to trace the sources of failing computations.

`+/-profile` (only available when configured during installation, see Section 1.1) Turn on/off the profile mode. If the profile mode is on, expressions as well as programs are compiled with GHC’s profiling capabilities enabled. For expressions, evaluation will automatically generate a file `Main.prof` containing the profiling information of the evaluation. For compiled programs, the profiling has to be manually activated using runtime options when executed:

```
kics2 :set +profile :load MyProgram.curry :save :quit
./MyProgram +RTS -p -RTS [additional arguments]
```

`+/-ghci` Turn on/off the ghci mode. In the ghci mode, the initial goal is send to the interactive version of the Glasgow Haskell Compiler. This might result in a slower execution but in a faster startup time since the linker to create the main executable is not used.

`safe` Turn on the safe execution mode. In the safe execution mode, the initial goal is not allowed to be of type `I0` and the program should not import the module `Unsafe`. Furthermore, the allowed commands are `eval`, `load`, `quit`, and `reload`. This mode is useful to use KiCS2 in uncontrolled environments, like a computation service in a web page, where KiCS2 could be invoked by

```
kics2 :set safe
```

`parser opts` Define additional options passed to the KiCS2 front end, i.e., the parser program `kics2home/bin/cymake`. For instance, setting the option

```
:set parser -F --pgmF=transcurry
```

has the effect that each Curry module to be compiled is transformed by the preprocessor command `transcurry` into a new Curry program which is actually compiled.

`cmp opts` Define additional options passed to the KiCS2 compiler. For instance, setting the option

```
:set cmp -O 0
```

has the effect that all optimizations performed by the KiCS2 compiler are turned off.

`ghc opts` Define additional options passed to the Glasgow Haskell Compiler (GHC) when the generated Haskell programs are compiled. Many options necessary to compile Curry programs are already set (you can see them by setting the verbosity level to 2 or greater). One has to be careful when providing additional options. For instance, in a global installation of KiCS2 (see Section 1.1), libraries are pre-compiled so that inconsistencies might occur if compilation options might be changed.

It is safe to pass specific GHC linking options. For instance, to enforce the static linking of libraries in order to generate an executable (see command “`:save`”) that can be executed in another environment, one could set the options

```
:set ghc -static -optl-static -optl-pthread
```

Other options are useful for experimental purposes, but those should be used only in local installations (see Section 1.1) to avoid inconsistent target codes for different libraries. For instance, setting the option

```
:set ghc -DDISABLE_CS
```

has the effect that the constraint store used to enable an efficient access to complex bindings is disabled. Similarly,

```
:set ghc -DSTRICT_VAL_BIND
```

has the effect that expressions in a unification constraint ($=:=$) are always fully evaluated (instead of the evaluation to a head normal form only) before unifying both sides. Since these options influence the compilation of the run-time system, one should also enforce the recompilation of Haskell programs by the GHC option “`-fforce-recomp`”, e.g., one should set

```
:set ghc -DDISABLE_CS -fforce-recomp
```

`rts opts` Define additional run-time options passed to the executable generated by the Glasgow Haskell Compiler, i.e., the parameters “`+RTS o -RTS`” are passed to the executable. For instance, setting the option

```
:set rts -H512m
```

has the effect that the minimum heap size is set to 512 megabytes.

`args arguments` Define run-time arguments passed to the executable generated by the Glasgow Haskell Compiler. For instance, setting the option

```
:set args first second
```

has the effect that the I/O operation `getArgs` (see library `System` (Section [A.2.40](#)) returns the value `["first","second"]`.

2.4 Source-File Options

If the evaluation of operations in some main module loaded into KiCS2 requires specific options, like an iterative-deepening search strategy, one can also put these options into the source code of this module in order to avoid setting these options every time when this module is loaded. Such **source-file options** must occur before the module header, i.e., before the first declaration (module header, imports, fixity declaration, defining rules, etc) occurring in the module. Each source file option must be in a line of the form

```
{-# KiCS2_OPTION opt #-}
```

where *opt* is an option that can occur in a “`:set`” command (compare Section [2.3](#)). Such a line in the source code (which is a comment according to the syntax of Curry) has the effect that this option is set by the KiCS2 command “`:set opt`” whenever this module is loaded (not reloaded!) as a main module. For instance, if a module starts with the lines

```
{-# KiCS2_OPTION ids #-}  
{-# KiCS2_OPTION +ghci #-}  
{-# KiCS2_OPTION v2 #-}  
module M where  
...  
...
```

then the load command “`:load M`” will also set the options for iterative deepening, using `ghci` and verbosity level 2.

2.5 Using KiCS2 in Batch Mode

Although KiCS2 is primarily designed as an interactive system, it can also be used to process data in batch mode. For example, consider a Curry program, say `myprocessor`, that reads argument strings from the command line and processes them. Suppose the entry point is a function called `just_doit` that takes no arguments. Such a processor can be invoked from the shell as follows:

```
> kics2 :set args string1 string2 :load myprocessor.curry :eval just_doit :quit
```

The “:quit” directive is necessary to avoid KiCS2 going into interactive mode after the execution of the expression being evaluated. The actual run-time arguments (`string1`, `string2`) are defined by setting the option `args` (see above).

Here is an example to use KiCS2 in this way:

```
> kics2 :set args Hello World :add System :eval "getArgs >>= putStrLn . unwords" :quit
Hello World
>
```

2.6 Command Line Editing

In order to have support for line editing or history functionality in the command line of KiCS2 (as often supported by the `readline` library), you should have the Unix command `rlwrap` installed on your local machine. If `rlwrap` is installed, it is used by KiCS2 if called on a terminal. If it should not be used (e.g., because it is executed in an editor with `readline` functionality), one can call KiCS2 with the parameter “`--noreadline`” (which must occur as the first parameter).

2.7 Customization

In order to customize the behavior of KiCS2 to your own preferences, there is a configuration file which is read by KiCS2 when it is invoked. When you start KiCS2 for the first time, a standard version of this configuration file is copied with the name “`.kics2rc`” into your home directory. The file contains definitions of various settings, e.g., about showing warnings, using Curry extensions, programs etc. After you have started KiCS2 for the first time, look into this file and adapt it to your own preferences.

2.8 Emacs Interface

Emacs is a powerful programmable editor suitable for program development. It is freely available for many platforms (see <http://www.emacs.org>). The distribution of KiCS2 contains also a special *Curry mode* that supports the development of Curry programs in the Emacs environment. This mode includes support for syntax highlighting, finding declarations in the current buffer, and loading Curry programs into KiCS2 in an Emacs shell.

The Curry mode has been adapted from a similar mode for Haskell programs. Its installation is described in the file `README` in directory “`kics2home/tools/emacs`” which also contains the sources of the Curry mode and a short description about the use of this mode.

3 Extensions

KiCS2 supports some extensions in Curry programs that are not (yet) part of the definition of Curry. These extensions are described below.

3.1 Narrowing on Int Literals

In addition to narrowing on algebraic data types, KiCS2 also implements narrowing on values of the primitive type `Int`. For example, the goal “`x == 3 where x free`” is evaluated to the solutions

```
Prelude> x == 3 where x free
{x = (-_x2)      } False
{x = 0           } False
{x = 1           } False
{x = (2 * _x3)   } False
{x = 3           } True
{x = (4 * _x4 + 1)} False
{x = (4 * _x4 + 3)} False
```

Note that the free variables occurring in the binding are restricted to positive numbers greater than 0 (the output has been indented to increase readability). This feature is implemented by an internal binary representation of integer numbers. If necessary, this representation can be exposed to the user by setting the flag `BinaryInt` during installation:

```
make [kernel|install] RUNTIMEFLAGS=BinaryInt
```

In an experimental (local) installation, the flag can also be set in the interpreter:

```
:set ghc -DBinaryInt
```

The example above will then be evaluated (without indentation) to:

```
Prelude> x == 3 where x free
{x = (Neg _x2)      } False
{x = 0             } False
{x = 1             } False
{x = (Pos (0 _x3))  } False
{x = 3             } True
{x = (Pos (I (0 _x4)))} False
{x = (Pos (I (I _x4)))} False
```

In this output, values without free variables are presented as before. For values containing a free variable, the constructors `Neg` and `Pos` denote negative and positive numbers (without 0), while the constructors `0` and `I` denote a 0- and 1-bit where the *least significant bit comes first*. That is, $(\text{Pos } (I (0 _x4))) = +(I (0 _x4)) = +(2 * (0 _x4)) + 1 = +(4 * _x4) + 1$ which meets the output above.

3.2 Recursive Variable Bindings

Local variable declarations (introduced by `let` or `where`) can be (mutually) recursive in KiCS2. For instance, the declaration

```
ones5 = let ones = 1 : ones
        in take 5 ones
```

introduces the local variable `ones` which is bound to a *cyclic structure* representing an infinite list of 1's. Similarly, the definition

```
onetwo n = take n one2
where
  one2 = 1 : two1
  two1 = 2 : one2
```

introduces a local variables `one2` that represents an infinite list of alternating 1's and 2's so that the expression `(onetwo 6)` evaluates to `[1,2,1,2,1,2]`.

3.3 Functional Patterns

Functional patterns [1] are a useful extension to code operations in a more readable way. Furthermore, defining operations with functional patterns avoids problems caused by strict equality (“`:=`”) and leads to programs that are potentially more efficient.

Consider the definition of an operation to compute the last element of a list `xs` based on the prelude operation “`++`” for list concatenation:

```
last xs | _++[y] := xs = y   where y free
```

Since the equality constraint “`:=`” evaluates both sides to a constructor term, all elements of the list `xs` are fully evaluated in order to satisfy the constraint.

Functional patterns can help to improve this computational behavior. A *functional pattern* is a function call at a pattern position. With functional patterns, we can define the operation `last` as follows:

```
last (_++[y]) = y
```

This definition is not only more compact but also avoids the complete evaluation of the list elements: since a functional pattern is considered as an abbreviation for the set of constructor terms obtained by all evaluations of the functional pattern to normal form (see [1] for an exact definition), the previous definition is conceptually equivalent to the set of rules

```
last [y] = y
last [_ , y] = y
last [_ , _ , y] = y
...
```

which shows that the evaluation of the list elements is not demanded by the functional pattern.

In general, a pattern of the form $(f\ t_1 \dots t_n)$ ($n > 0$) is interpreted as a functional pattern if f is not a visible constructor but a defined function that is visible in the scope of the pattern.

It is also possible to combine functional patterns with as-patterns. Similarly to the meaning of as-patterns in standard constructor patterns, as-patterns in functional patterns are interpreted as a sequence of pattern matching where the variable of the as-pattern is matched before the given pattern is matched. This process can be described by introducing an auxiliary operation for this two-level pattern matching process. For instance, the definition

```
f (_ ++ x@[42,_] ++ _) = x
```

is considered as syntactic sugar for the expanded definition

```
f (_ ++ x ++ _) = f' x
  where
    f' [(42,_)] = x
```

However, as-patterns are usually implemented in a more efficient way without introducing auxiliary operations.

3.4 Order of Pattern Matching

Curry allows multiple occurrences of pattern variables in standard patterns. These are an abbreviation of equational constraints between pattern variables. Functional patterns might also contain multiple occurrences of pattern variables. For instance, the operation

```
f (_++[x]++_++[x]++) = x
```

returns all elements with at least two occurrences in a list.

If functional patterns as well as multiple occurrences of pattern variables occur in a pattern defining an operation, there are various orders to match an expression against such an operation. In the current implementation, the order is as follows:

1. Standard pattern matching: First, it is checked whether the constructor patterns match. Thus, functional patterns and multiple occurrences of pattern variables are ignored.
2. Functional pattern matching: In the next phase, functional patterns are matched but occurrences of standard pattern variables in the functional patterns are ignored.
3. Non-linear patterns: If standard and functional pattern matching is successful, the equational constraints which correspond to multiple occurrences pattern variables are solved.
4. Guards: Finally, the guards supplied by the programmer are checked.

The order of pattern matching should not influence the computed result. However, it might have some influence on the termination behavior of programs, i.e., a program might not terminate instead of finitely failing. In such cases, it could be necessary to consider the influence of the order of pattern matching. Note that other orders of pattern matching can be obtained using auxiliary operations.

4 Recognized Syntax of Curry

The KiCS2 Curry compiler accepts a slightly extended version of the grammar specified in the Curry Report [20]. Furthermore, the syntax recognized by KiCS2 differs from that specified in the Curry Report regarding numeric or character literals. We therefore present the complete description of the syntax below, whereas *syntactic extensions* are highlighted.

4.1 Notational Conventions

The syntax is given in extended Backus-Naur-Form (eBNF), using the following notation:

<i>NonTerm</i>	::= α	production
<i>NonTerm</i>		nonterminal symbol
Term		terminal symbol
	$[\alpha]$	optional
	$\{\alpha\}$	zero or more repetitions
	(α)	grouping
	$\alpha \mid \beta$	alternative
	$\alpha_{\langle\beta\rangle}$	difference – elements generated by α without those generated by β

The Curry files are expected to be encoded in UTF8. However, source programs are biased towards ASCII for compatibility reasons.

4.2 Lexicon

4.2.1 Case Mode

Although the Curry Report specifies four different case modes (Prolog, Gödel, Haskell, free), the KiCS2 only supports the *free* mode which puts no constraints on the case of identifiers.

4.2.2 Identifiers and Keywords

<i>Letter</i>	::=	any ASCII letter
<i>Dashes</i>	::=	-- {-}
<i>Ident</i>	::=	<i>Letter</i> { <i>Letter</i> <i>Digit</i> $_$ $'$ }
<i>Symbol</i>	::=	\sim $!$ $@$ $\#$ $\$$ $\%$ \wedge $\&$ $*$ $+$ $-$ $=$ $<$ $>$ $?$ $.$ $/$ $ $ \backslash $:$
<i>ModuleID</i>	::=	{ <i>Ident</i> .} <i>Ident</i>
<i>TypeConstrID</i>	::=	<i>Ident</i>
<i>DataConstrID</i>	::=	<i>Ident</i>
<i>TypeVarID</i>	::=	<i>Ident</i> $_$
<i>InfixOpID</i>	::=	(<i>Symbol</i> { <i>Symbol</i> }) _{<i>Dashes</i>}
<i>FunctionID</i>	::=	<i>Ident</i>
<i>VariableID</i>	::=	<i>Ident</i>
<i>LabelID</i>	::=	<i>Ident</i>

```

QTypeConstrID ::= [ModuleID .] TypeConstrID
QDataConstrID ::= [ModuleID .] DataConstrID
  QInfixOpID ::= [ModuleID .] InfixOpID
QFunctionID ::= [ModuleID .] FunctionID
QVariableID ::= [ModuleID .] VariableID
  QLabelID ::= [ModuleID .] LabelID

```

The following identifiers are recognized as keywords and cannot be used as an identifier:

```

case      data      do      else      external  fcase      foreign
free      if        import   in        infix     infixl     infixr
let       module    newtype  of       then      type       where

```

Note that the symbols `as`, `hiding` and `qualified` are not keywords. They have only a special meaning in module headers and can be used as ordinary identifiers.

The following symbols also have a special meaning and cannot be used as an infix operator identifier:

```

..      :      ::     =      \      |      <-     ->     @      ~

```

4.2.3 Comments

Comments begin either with “`--`” and terminate at the end of the line or with “`{-`” and terminate with a matching “`-}`”, i.e., the delimiters “`{-`” and “`-}`” act as parentheses and can be nested.

4.2.4 Numeric and Character Literals

Contrasting to the Curry Report, KiCS2 adopts Haskell’s notation of literals, for both numeric literals as well as `Char` and `String` literals. The precise syntax for both kinds is given below.

```

Int ::= Decimal
      | 0b Binary | 0B Binary
      | 0o Octal | 0O Octal
      | 0x Hexadecimal | 0X Hexadecimal

Float ::= Decimal . Decimal [Exponent]
          | Decimal Exponent
Exponent ::= (e | E) [+ | -] Decimal

Decimal ::= Digit [Decimal]
Binary ::= Binit [Binary]
Octal ::= Octit [Octal]
Hexadecimal ::= Hexit [Hexadecimal]

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Binit ::= 0 | 1
Octit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Hexit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

Char ::= '(Graphic1 | Space | Escape(\&))'
String ::= "{Graphicn | Space | Escape | Gap}"

```

```

Escape ::= \ (CharEsc | Ascii | Decimal | o Octal | x Heradecimal)
CharEsc ::= a | b | f | n | r | t | v | \ | " | ' | &
Ascii ::= ^ Cntrl | NUL | SOH | STX | ETX | EOT | ENQ | ACK
        | BEL | BS | HT | LF | VT | FF | CR | SO | SI | DLE
        | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN
        | EM | SUB | ESC | FS | GS | RS | US | SP | DEL
Cntrl ::= AsciiLarge | @ | [ | \ | ] | ^ | _
AsciiLarge ::= A | ... | Z
Gap ::= \ WhiteChar { WhiteChar } \

```

4.3 Layout

Similarly to Haskell, a Curry programmer can use layout information to define the structure of blocks. For this purpose, we define the indentation of a symbol as the column number indicating the start of this symbol. The indentation of a line is the indentation of its first symbol.²

The layout (or “off-side”) rule applies to lists of syntactic entities after the keywords **let**, **where**, **do**, or **of**. In the subsequent context-free syntax, these lists are enclosed with curly brackets (`{ }`) and the single entities are separated by semicolons (`;`). Instead of using the curly brackets and semicolons of the context-free syntax, a Curry programmer must specify these lists by indentation: the indentation of a list of syntactic entities after **let**, **where**, **do**, or **of** is the indentation of the next symbol following the **let**, **where**, **do**, **of**. Any item of this list start with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in its previous line. Lines with an indentation less than the indentation of the list terminate the entire list. Moreover, a list started by **let** is terminated by the keyword **in**. Thus, the sentence

```
f x = h x where { g y = y + 1 ; h z = (g z) * 2 }
```

which is valid w.r.t. the context-free syntax, is written with the layout rules as

```
f x = h x
  where g y = y + 1
        h z = (g z) * 2
```

or also as

```
f x = h x  where
  g y = y + 1
  h z = (g z)
        * 2
```

To avoid an indentation of top-level declarations, the keyword **module** and the end-of-file token are assumed to start in column 0.

²In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.

4.4 Context Free Grammar

<i>Module</i>	::=	module <i>ModuleID</i> [<i>Exports</i>] where <i>Block</i>	
		<i>Block</i>	
<i>ModuleID</i>	::=	see lexicon	
<i>Exports</i>	::=	(<i>Export</i> ₁ , ... , <i>Export</i> _{<i>n</i>})	(<i>n</i> ≥ 0)
<i>Export</i>	::=	<i>QFunctionName</i>	
		<i>QTypeConstrID</i> [(<i>ConsLabelName</i> ₁ , ... , <i>ConsLabelName</i> _{<i>n</i>})]	(<i>n</i> ≥ 0)
		<i>QTypeConstrID</i> (...)	
		module <i>ModuleID</i>	
<i>ConsLabelName</i>	::=	<i>LabelID</i> <i>DataConstr</i>	
<i>Block</i>	::=	{ [<i>ImportDecl</i> ₁ ; ... ; <i>ImportDecl</i> _{<i>k</i>} ;]	(<i>no fixity declarations here</i>)
		<i>BlockDeclaration</i> ₁ ; ... ; <i>BlockDeclaration</i> _{<i>n</i>} }	(<i>k, n</i> ≥ 0)
<i>ImportDecl</i>	::=	import [qualified] <i>ModuleID</i> [as <i>ModuleID</i>] [<i>ImportRestr</i>]	
<i>ImportRestr</i>	::=	(<i>Import</i> ₁ , ... , <i>Import</i> _{<i>n</i>})	(<i>n</i> ≥ 0)
		hiding (<i>Import</i> ₁ , ... , <i>Import</i> _{<i>n</i>})	(<i>n</i> ≥ 0)
<i>Import</i>	::=	<i>FunctionName</i>	
		<i>TypeConstrID</i> [(<i>ConsLabelName</i> ₁ , ... , <i>ConsLabelName</i> _{<i>n</i>})]	(<i>n</i> ≥ 0)
		<i>TypeConstrID</i> (...)	
<i>BlockDeclaration</i>	::=	<i>TypeSynonymDecl</i>	
		<i>DataDeclaration</i>	
		<i>FixityDeclaration</i>	
		<i>FunctionDeclaration</i>	
<i>TypeSynonymDecl</i>	::=	type <i>SimpleType</i> = <i>TypeExpr</i>	
<i>SimpleType</i>	::=	<i>TypeConstrID</i> <i>TypeVarID</i> ₁ ... <i>TypeVarID</i> _{<i>n</i>}	(<i>n</i> ≥ 0)
<i>TypeConstrID</i>	::=	see lexicon	
<i>DataDeclaration</i>	::=	data <i>SimpleType</i>	(<i>external data type</i>)
		data <i>SimpleType</i> = <i>ConstrDecl</i> ₁ ... <i>ConstrDecl</i> _{<i>n</i>}	(<i>n</i> > 0)
<i>ConstrDecl</i>	::=	<i>DataConstr</i> <i>SimpleTypeExpr</i> ₁ ... <i>SimpleTypeExpr</i> _{<i>n</i>}	(<i>n</i> ≥ 0)
		<i>SimpleTypeExpr</i> <i>ConsOp</i> <i>TypeConsExpr</i>	(<i>infix data constructor</i>)
		<i>DataConstr</i> { <i>FieldDeclaration</i> ₁ , ... , <i>FieldDeclaration</i> _{<i>n</i>} }	(<i>n</i> ≥ 0)
<i>FieldDeclaration</i>	::=	<i>LabelID</i> ₁ , ... , <i>LabelID</i> _{<i>n</i>} :: <i>TypeExpr</i>	(<i>n</i> > 0)
<i>LabelID</i>	::=	see lexicon	
<i>TypeExpr</i>	::=	<i>TypeConsExpr</i> [-> <i>TypeExpr</i>]	
<i>TypeConsExpr</i>	::=	<i>QTypeConstrID</i> <i>SimpleTypeExpr</i> ₁ ... <i>SimpleTypeExpr</i> _{<i>n</i>}	(<i>n</i> > 0)
		<i>SimpleTypeExpr</i>	
<i>SimpleTypeExpr</i>	::=	<i>TypeVarID</i>	
		<i>QTypeConstrID</i>	
		()	(<i>unit type</i>)
		(<i>TypeExpr</i> ₁ , ... , <i>TypeExpr</i> _{<i>n</i>})	(<i>tuple type, n</i> > 1)
		[<i>TypeExpr</i>]	(<i>list type</i>)
		(<i>TypeExpr</i>)	(<i>parenthesized type</i>)

TypeVarID ::= see lexicon
FixityDeclaration ::= *FixityKeyword* *Digit* *InfixOpID*₁ , ... , *InfixOpID*_n (n > 0)
FixityKeyword ::= infixl | infixr | infix
InfixOpID ::= see lexicon
FunctionDeclaration ::= *Signature* | *External* | *Equat*
External ::= *FunctionNames* *external* (externally defined functions)
Signature ::= *FunctionNames* :: *TypeExpr*
FunctionNames ::= *FunctionName*₁ , ... , *FunctionName*_n (n > 0)
Equat ::= *FunLHS* = *Expr* [where *LocalDefs*]
| *FunLHS* *CondExprs* [where *LocalDefs*]
FunLHS ::= *FunctionName* *SimplePat*₁ ... *SimplePat*_n (n ≥ 0)
| *SimplePat* *InfixOpID* *SimplePat*
CondExprs ::= | *InfixExpr* = *Expr* [*CondExprs*]
Pattern ::= *ConsPattern* [*QConsOp* *Pattern*] (infix constructor pattern)
ConsPattern ::= *GDataConstr* *SimplePat*₁ ... *SimplePat*_n (constructor pattern)
| *SimplePat*
SimplePat ::= *Variable* (wildcard)
| -
| *QDataConstr*
| *Literal*
| - *Int* (negative pattern)
| - . *Float* (negative float pattern)
| () (empty tuple pattern)
| (*Pattern*₁ , ... , *Pattern*_n) (n > 1)
| (*Pattern*) (parenthesized pattern)
| [*Pattern*₁ , ... , *Pattern*_n] (n ≥ 0)
| *Variable* @ *SimplePat* (as-pattern)
| ~ *SimplePat* (irrefutable pattern)
| (*SimplePat* *QFunOp* *SimplePat*) (infix functional pattern)
| (*QFunctionName* *SimplePat*₁ ... *SimplePat*_n) (functional pattern, n > 0)
| *QDataConstr* { *FieldPat*₁ , ... , *FieldPat*_n } (labeled pattern, n ≥ 0)
FieldPat ::= *QLabelID* = *Pattern*
QLabelID ::= see lexicon
LocalDefs ::= { *ValueDeclaration*₁ ; ... ; *ValueDeclaration*_n } (n > 0)
ValueDeclaration ::= *FunctionDeclaration*
| *PatternDeclaration*
| *VariableID*₁ , ... , *VariableID*_n **free** (n > 0)
| *FixityDeclaration*
PatternDeclaration ::= *Pattern* = *Expr* [where *LocalDefs*]
Expr ::= *InfixExpr* :: *TypeExpr* (expression type signature)

		<i>InfixExpr</i>	
<i>InfixExpr</i>	::=	<i>NoOpExpr</i> <i>QOp</i> <i>InfixExpr</i>	(infix operator application)
		- <i>InfixExpr</i>	(unary int minus)
		- . <i>InfixExpr</i>	(unary float minus)
		<i>NoOpExpr</i>	
<i>NoOpExpr</i>	::=	\ <i>SimplePat</i> ₁ ... <i>SimplePat</i> _n -> <i>Expr</i>	(lambda expression, n > 0)
		let <i>LocalDefs</i> in <i>Expr</i>	(let expression)
		if <i>Expr</i> then <i>Expr</i> else <i>Expr</i>	(conditional)
		case <i>Expr</i> of { <i>Alt</i> ₁ ; ... ; <i>Alt</i> _n }	(case expression, n ≥ 0)
		fcase <i>Expr</i> of { <i>Alt</i> ₁ ; ... ; <i>Alt</i> _n }	(fcase expression, n ≥ 0)
		do { <i>Stmt</i> ₁ ; ... ; <i>Stmt</i> _n ; <i>Expr</i> }	(do expression, n ≥ 0)
		<i>FunctExpr</i>	
<i>FunctExpr</i>	::=	[<i>FunctExpr</i>] <i>BasicExpr</i>	(function application)
<i>BasicExpr</i>	::=	<i>QVariableID</i>	(variable)
		-	(anonymous free variable)
		<i>QFunctionName</i>	(qualified function)
		<i>GDataConstr</i>	(general constructor)
		<i>Literal</i>	
		(<i>Expr</i>)	(parenthesized expression)
		(<i>Expr</i> ₁ , ... , <i>Expr</i> _n)	(tuple, n > 1)
		[<i>Expr</i> ₁ , ... , <i>Expr</i> _n]	(finite list, n > 0)
		[<i>Expr</i> [, <i>Expr</i>] .. [<i>Expr</i>]]	(arithmetic sequence)
		[<i>Expr</i> <i>Qual</i> ₁ , ... , <i>Qual</i> _n]	(list comprehension, n ≥ 1)
		(<i>InfixExpr</i> <i>QOp</i>)	(left section)
		(<i>QOp</i> _(-, -, .) <i>InfixExpr</i>)	(right section)
		<i>QDataConstr</i> { <i>FBind</i> ₁ , ... , <i>FBind</i> _n }	(labeled construction, n ≥ 0)
		<i>BasicExpr</i> _(<i>QDataConstr</i>) { <i>FBind</i> ₁ , ... , <i>FBind</i> _n }	(labeled update, n > 0)
<i>Alt</i>	::=	<i>Pattern</i> -> <i>Expr</i> [where <i>LocalDefs</i>]	
		<i>Pattern</i> <i>GdAlts</i> [where <i>LocalDefs</i>]	
<i>GdAlts</i>	::=	<i>InfixExpr</i> -> <i>Expr</i> [<i>GdAlts</i>]	
<i>FBind</i>	::=	<i>QLabelID</i> = <i>Expr</i>	
<i>Qual</i>	::=	<i>Expr</i>	
		let <i>LocalDefs</i>	
		<i>Pattern</i> <- <i>Expr</i>	
<i>Stmt</i>	::=	<i>Expr</i>	
		let <i>LocalDefs</i>	
		<i>Pattern</i> <- <i>Expr</i>	
<i>Literal</i>	::=	<i>Int</i> <i>Char</i> <i>String</i> <i>Float</i>	
<i>GDataConstr</i>	::=	()	
		[]	
		(, {, })	
		<i>QDataConstr</i>	
<i>FunctionName</i>	::=	<i>FunctionID</i> (<i>InfixOpID</i>)	(function)
<i>QFunctionName</i>	::=	<i>QFunctionID</i> (<i>QInfixOpID</i>)	(qualified function)

$Variable$	$::= VariableID \mid (InfixOpID)$	$(variable)$
$DataConstr$	$::= DataConstrID \mid (InfixOpID)$	$(constructor)$
$QDataConstr$	$::= QDataConstrID \mid (QConsOp)$	$(qualified\ constructor)$
$QFunOp$	$::= QInfixOpID \mid 'QFunctionID'$	$(qualified\ function\ operator)$
$ConsOp$	$::= InfixOpID \mid 'DataConstrID'$	$(constructor\ operator)$
QOp	$::= QFunOp \mid QConsOp$	$(qualified\ operator)$
$QConsOp$	$::= GConSym \mid 'QDataConstrID'$	$(qualified\ constructor\ operator)$
$GConSym$	$::= : \mid QInfixOpID$	$(general\ constructor\ symbol)$

5 Optimization of Curry Programs

After the invocation of the Curry front end, which parses a Curry program and translates it into the intermediate FlatCurry representation, KiCS2 applies a transformation to optimize Boolean equalities occurring in the Curry program. The ideas and details of this optimization are described in [4]. Therefore, we sketch only some basic ideas and options to influence this optimization.

Consider the following definition of the operation `last` to extract the last element in list:

```
last xs | xs == _++[x]
      = x
where x free
```

In order to evaluate the condition “`xs == _++[x]`”, the Boolean equality is evaluated to `True` or `False` by instantiating the free variables `_` and `x`. However, since we know that a condition must be evaluated to `True` only and all evaluations to `False` can be ignored, we can use the constrained equality to obtain a more efficient program:

```
last xs | xs :=: _++[x]
      = x
where x free
```

Since the selection of the appropriate equality operator is not obvious and might be tedious, KiCS2 encourages programmers to use only the Boolean equality operator “`==`” in programs. The constraint equality operator “`:=:`” can be considered as an optimization of “`==`” if it is ensured that only positive results are required, e.g., in conditions of program rules.

To support this programming style, KiCS2 has a built-in optimization phase on FlatCurry files. For this purpose, the optimizer analyzes the FlatCurry programs for occurrences of “`==`” and replaces them by “`:=:`” whenever the result `False` is not required. The usage of the optimizer can be influenced by setting the property flag `bindingoptimization` in the configuration file `.kics2rc`. The following values are recognized for this flag:

no: Do not apply this transformation.

fast: This is the default value. The transformation is based on pre-computed values for the prelude operations in order to decide whether the value `False` is not required as a result of a Boolean equality. Hence, the transformation can be efficiently performed without any complex analysis.

full: Perform a complete “required values” analysis of the program (see [4]) and use this information to optimize programs. In most cases, this does not yield better results so that the `fast` mode is sufficient.

Hence, to turn off this optimization, one can either modify the flag `bindingoptimization` in the configuration file `.kics2rc` or dynamically pass this change to the invocation of KiCS2 by

```
... -Dbindingoptimization=no ...
```

6 CurryDoc: A Documentation Generator for Curry Programs

CurryDoc is a tool in the KiCS2 distribution that generates the documentation for a Curry program (i.e., the main module and all its imported modules) in HTML format. The generated HTML pages contain information about all data types and functions exported by a module as well as links between the different entities. Furthermore, some information about the definitional status of functions (like rigid, flexible, external, complete, or overlapping definitions) are provided and combined with documentation comments provided by the programmer.

A *documentation comment* starts at the beginning of a line with “`---` ” (also in literate programs!). All documentation comments immediately before a definition of a datatype or (top-level) function are kept together.³ The documentation comments for the complete module occur before the first “`module`” or “`import`” line in the module. The comments can also contain several special tags. These tags must be the first thing on its line (in the documentation comment) and continues until the next tag is encountered or until the end of the comment. The following tags are recognized:

`@author` *comment*

Specifies the author of a module (only reasonable in module comments).

`@version` *comment*

Specifies the version of a module (only reasonable in module comments).

`@cons` *id comment*

A comment for the constructor *id* of a datatype (only reasonable in datatype comments).

`@param` *id comment*

A comment for function parameter *id* (only reasonable in function comments). Due to pattern matching, this need not be the name of a parameter given in the declaration of the function but all parameters for this functions must be commented in left-to-right order (if they are commented at all).

`@return` *comment*

A comment for the return value of a function (only reasonable in function comments).

The comment of a documented entity can be any string in [Markdown’s syntax](#) (the currently supported set of elements is described in detail in the appendix). For instance, it can contain Markdown annotations for emphasizing elements (e.g., `_verb_`), strong elements (e.g., `**important**`), code elements (e.g., `‘3+4’`), code blocks (lines prefixed by four blanks), unordered lists (lines prefixed by “`*`”), ordered lists (lines prefixed by blanks followed by a digit and a dot), quotations (lines prefixed by “`>`”), and web links of the form “`<http://...>`” or “`[link text](http://...)`”. If the Markdown syntax should not be used, one could run CurryDoc with the parameter “`--nomarkdown`”.

The comments can also contain markups in HTML format so that special characters like “`<`” must be quoted (e.g., “`<`”). However, header tags like `<h1>` should not be used since the structuring is generated by CurryDoc. In addition to Markdown or HTML markups, one can also mark *references to names* of operations or data types in Curry programs which are translated into links

³The documentation tool recognizes this association from the first identifier in a program line. If one wants to add a documentation comment to the definition of a function which is an infix operator, the first line of the operator definition should be a type definition, otherwise the documentation comment is not recognized.

inside the generated HTML documentation. Such references have to be enclosed in single quotes. For instance, the text `'conc'` refers to the Curry operation `conc` inside the current module whereas the text `'Prelude.reverse'` refers to the operation `reverse` of the module `Prelude`. If one wants to write single quotes without this specific meaning, one can escape them with a backslash:

```
--- This is a comment without a \'reference\'.

```

To simplify the writing of documentation comments, such escaping is only necessary for single words, i.e., if the text inside quotes has not the syntax of an identifier, the escaping can be omitted, as in

```
--- This isn't a reference.

```

The following example text shows a Curry program with some documentation comments:

```
--- This is an
--- example module.
--- @author Michael Hanus
--- @version 0.1

module Example where

--- The function 'conc' concatenates two lists.
--- @param xs - the first list
--- @param ys - the second list
--- @return a list containing all elements of 'xs' and 'ys'
conc []      ys = ys
conc (x:xs) ys = x : conc xs ys
-- this comment will not be included in the documentation

--- The function 'last' computes the last element of a given list.
--- It is based on the operation 'conc' to concatenate two lists.
--- @param xs - the given input list
--- @return last element of the input list
last xs | conc ys [x] ::= xs = x   where x,ys free

--- This data type defines _polymorphic_ trees.
--- @cons Leaf - a leaf of the tree
--- @cons Node - an inner node of the tree
data Tree a = Leaf a | Node [Tree a]

```

To generate the documentation, execute the command

```
currydoc Example

```

(`currydoc` is a command usually stored in `kics2home/bin` (where `kics2home` is the installation directory of KiCS2; see Section 1.2). This command creates the directory `DOC_Example` (if it does not exist) and puts all HTML documentation files for the main program module `Example` and all its imported modules in this directory together with a main index file `index.html`. If one prefers another directory for the documentation files, one can also execute the command

```
currydoc docdir Example

```

where `docdir` is the directory for the documentation files.

In order to generate the common documentation for large collections of Curry modules (e.g., the libraries contained in the KiCS2 distribution), one can call `currydoc` with the following options:

`currydoc --noindexhtml docdir Mod` : This command generates the documentation for module `Mod` in the directory `docdir` without the index pages (i.e., main index page and index pages for all functions and constructors defined in `Mod` and its imported modules).

`currydoc --onlyindexhtml docdir Mod1 Mod2 ...Modn` : This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules `Mod1`, `M2`, ..., `Modn` and their imported modules) in the directory `docdir`.

7 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs

CurryBrowser is a tool to browse through the modules and functions of a Curry application, show them in various formats, and analyze their properties.⁴ Moreover, it is constructed in a way so that new analyzers can be easily connected to CurryBrowser. A detailed description of the ideas behind this tool can be found in [15, 16].

CurryBrowser is part of the KiCS2 distribution and can be started in two ways:

- In the command shell via the command: `kics2home/bin/currybrowser mod`
- In the KiCS2 environment after loading the module `mod` and typing the command “:browse”.

Here, “mod” is the name of the main module of a Curry application. After the start, CurryBrowser loads the interfaces of the main module and all imported modules before a GUI is created for interactive browsing.

To get an impression of the use of CurryBrowser, Figure 1 shows a snapshot of its use on a particular application (here: the implementation of CurryBrowser). The upper list box in the left column shows the modules and their imports in order to browse through the modules of an application. Similarly to directory browsers, the list of imported modules of a module can be opened or closed by clicking. After selecting a module in the list of modules, its source code, interface, or various other formats of the module can be shown in the main (right) text area. For instance, one can show pretty-printed versions of the intermediate flat programs (see below) in order to see how local function definitions are translated by lambda lifting [21] or pattern matching is translated into case expressions [11, 22]. Since Curry is a language with parametric polymorphism and type inference, programmers often omit the type signatures when defining functions. Therefore, one can also view (and store) the selected module as source code where missing type signatures are added.

Below the list box for selecting modules, there is a menu (“Analyze selected module”) to analyze all functions of the currently selected module at once. This is useful to spot some functions of a module that could be problematic in some application contexts, like functions that are impure (i.e., the result depends on the evaluation time) or partially defined (i.e., not evaluable on all ground terms). If such an analysis is selected, the names of all functions are shown in the lower list box of the left column (the “function list”) with prefixes indicating the properties of the individual functions.

The function list box can be also filled with functions via the menu “Select functions”. For instance, all functions or only the exported functions defined in the currently selected module can be shown there, or all functions from different modules that are directly or indirectly called from a currently selected function. This list box is central to focus on a function in the source code of some module or to analyze some function, i.e., showing their properties. In order to focus on a function, it is sufficient to check the “focus on code” button. To analyze an individually selected function, one can select an analysis from the list of available program analyses (through the menu “Select analysis”). In this case, the analysis results are either shown in the text box below the main text area or visualized by separate tools, e.g., by a graph drawing tool for visualizing call graphs. Some

⁴Although CurryBrowser is implemented in Curry, some functionalities of it require an installed graph visualization tool (dot <http://www.graphviz.org/>), otherwise they have no effect.

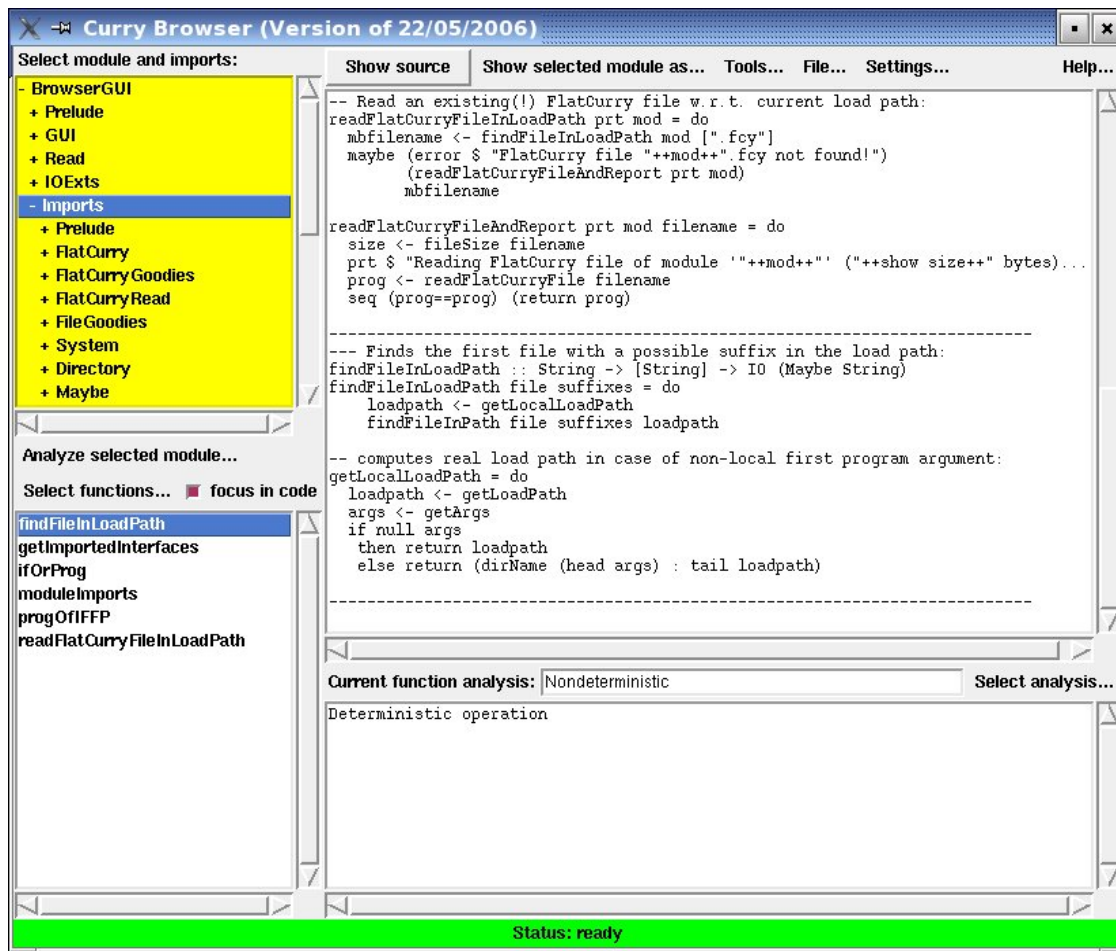


Figure 1: Snapshot of the main window of CurryBrowser

analyses are local, i.e., they need only to consider the local definition of this function (e.g., “Calls directly,” “Overlapping rules,” “Pattern completeness”), where other analyses are global, i.e., they consider the definitions of all functions directly or indirectly called by this function (e.g., “Depends on,” “Solution complete,” “Set-valued”). Finally, there are a few additional tools integrated into CurryBrowser, for instance, to visualize the import relation between all modules as a dependency graph. These tools are available through the “Tools” menu.

More details about the use of CurryBrowser and all built-in analyses are available through the “Help” menu of CurryBrowser.

8 CurryTest: A Tool for Testing Curry Programs

CurryTest is a simple tool in the KiCS2 distribution to write and run repeatable tests. CurryTest simplifies the task of writing test cases for a module and executing them. The tool is easy to use. Assume one has implemented a module `MyMod` and wants to write some test cases to test its functionality, making regression tests in future versions, etc. For this purpose, there is a system library `Assertion` (Section A.2.2) which contains the necessary definitions for writing tests. In particular, it exports an abstract polymorphic type “`Assertion a`” together with the following operations:

```
assertTrue      :: String → Bool → Assertion ()
assertEqual     :: String → a → a → Assertion a
assertValues    :: String → a → [a] → Assertion a
assertSolutions :: String → (a → Bool) → [a] → Assertion a
assertIO        :: String → IO a → a → Assertion a
assertEqualIO   :: String → IO a → IO a → Assertion a
```

The expression “`assertTrue s b`” is an assertion (named `s`) that the expression `b` has the value `True`. Similarly, the expression “`assertEqual s e1 e2`” asserts that the expressions `e1` and `e2` must be equal (i.e., `e1==e2` must hold), the expression “`assertValues s e vs`” asserts that `vs` is the multiset of all values of `e`, and the expression “`assertSolutions s c vs`” asserts that the constraint abstraction `c` has the multiset of solutions `vs`. Furthermore, the expression “`assertIO s a v`” asserts that the I/O action `a` yields the value `v` whenever it is executed, and the expression “`assertEqualIO s a1 a2`” asserts that the I/O actions `a1` and `a2` yield equal values. The name `s` provided as a first argument in each assertion is used in the protocol produced by the test tool.

One can define a test program by importing the module to be tested together with the module `Assertion` and defining top-level functions of type `Assertion` in this module (which must also be exported). As an example, consider the following program that can be used to test some list processing functions:

```
import List
import Assertion

test1 = assertEqual    "++"    ([1,2]++[3,4]) [1,2,3,4]

test2 = assertTrue     "all"    (all (<5) [1,2,3,4])

test3 = assertSolutions "prefix" (\x → x++_ := [1,2])
                                     [[], [1], [1,2]]
```

For instance, `test1` asserts that the result of evaluating the expression `([1,2]++[3,4])` is equal to `[1,2,3,4]`.

We can execute a test suite by the command

```
currytest TestList
```

(`currytest` is a program stored in `kics2home/bin` where `kics2home` is the installation directory of KiCS2; see Section 1.2). In our example, “`TestList.curry`” is the program containing the definition of all assertions. This has the effect that all exported top-level functions of type `Assertion` are

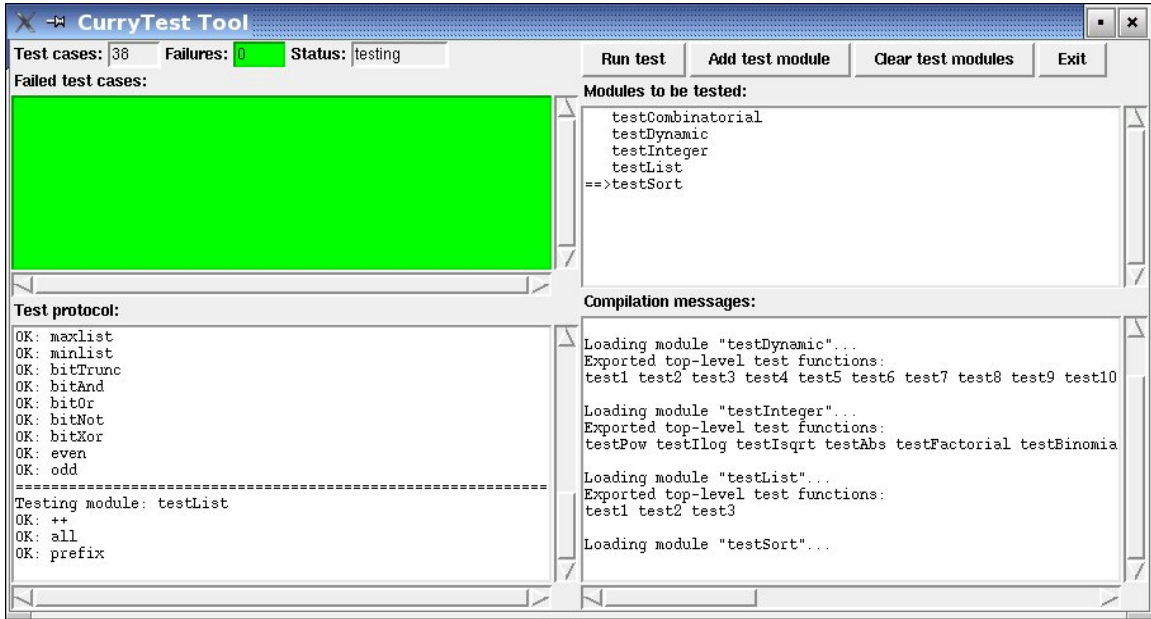


Figure 2: Snapshot of CurryTest’s graphical interface

tested (i.e., the corresponding assertions are checked) and the results (“OK” or failure) are reported together with the name of each assertion. For our example above, we obtain the following successful protocol:

```

=====
Testing module "TestList"...
OK: ++
OK: all
OK: prefix
All tests successfully passed.
=====

```

There is also a graphical interface that summarizes the results more nicely. In order to start this interface, one has to add the parameter “--window” (or “-w”), e.g., executing a test suite by

```
currytest --window TestList
```

or

```
currytest -w TestList
```

A snapshot of the interface is shown in Figure 2.

9 CurryPP: A Preprocessor for Curry Programs

The Curry preprocessor “`currypp`” implements various transformations on Curry source programs. It supports some experimental language extensions that might become part of the standard parser of Curry in some future version.

Currently, the Curry preprocessor supports the following extensions that will be described below in more detail:

Integrated code: This extension allows to integrate code written in some other language into Curry programs, like regular expressions, format specifications (“`printf`”), HTML and XML code.

Sequential rules: If this feature is used, all rules in a Curry module are interpreted as sequential, i.e., a rule is applied only if all previous rules defining the same operation are not applicable. The idea of sequential rules are described in [3].

Default rules: If this feature is used, one can add a default rule to operations defined in a Curry module. This provides a similar power than sequential rules but with a better operational behavior. The idea of default rules are described in [5].

The preprocessor is an executable named “`currypp`”, which is stored in the directory `kics2home/bin`. In order to apply the preprocessor when loading a Curry source program into KiCS2, one has to add an option line at the beginning of the source program. For instance, in order to use default rules in a Curry program, one has to put the line

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}
```

at the beginning of the program. This option tells the KiCS2 front end to process the Curry source program with `currypp` before actually parsing the source text.

The option “`defaultrules`” has to be replaced by “`seqrules`” if the sequential rule matching should be replaced. To support integrated code, one has to set the option “`foreigncode`” (which can also be combined with either “`defaultrules`” or “`seqrules`”). If one wants to see the result of the transformation, one can also set the option “`-o`”. This has the effect that the transformed source program is stored in the file `Prog.curry.CURRYPP` if the name of the original program is `Prog.curry`.

For instance, in order to use integrated code and default rules in a module and store the transformed program, one has to put the line

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=-o #-}
```

at the beginning of the program.

9.1 Integrated Code

Integrated code is enclosed in at least two back ticks and ticks in a Curry program. The number of starting back ticks and ending ticks must be always identical. After the initial back ticks, there must be an identifier specifying the kind of integrated code, e.g., `regexp` or `html` (see below). For instance, if one uses regular expressions (see below for more details), the following expressions are valid in source programs:

```
s 'regex (a|(bc*))+'
s '“““regex aba*c”””’
```

The Curry preprocessor transforms these code pieces into regular Curry expressions. The next sections describe the currently supported foreign languages.

9.1.1 Regular Expressions

In order to match strings against regular expressions, i.e., to check whether a string is contained in the language generated by a regular expression, one can specify regular expression similar to POSIX. The foreign regular expression code must be marked by “`regex`”. Since this code is transformed into operations of the KiCS2 library `RegExp`, this library must be imported.

For instance, the following module defines a predicate to check whether a string is a valid identifier:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import RegExp

isID :: String → Bool
isID s = s 'regex [a-zA-Z][a-zA-Z0-9_]*'
```

9.1.2 Format Specifications

In order to format numerical and other data as strings, one can specify the desired format with foreign code marked by “`format`”. In this case, one can write a format specification, similarly to the `printf` statement of C, followed by a comma-separated list of arguments. This format specification is transformed into operations of the KiCS2 library `Format` so that it must be imported. For instance, the following program defines an operation that formats a string, an integer (with leading sign and zeros), and a float with leading sign and precision 3:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import Format

showSIF :: String → Int → Float → String
showSIF s i f = 'format "Name: %s | %+.5i | %+6.3f",s,i,f'

main = putStrLn $ showSIF "Curry" 42 3.14159
```

Thus, the execution of `main` will print the line

```
Name: Curry | +00042 | +3.142
```

Instead of “`format`”, one can also write a format specification with `printf`. In this case, the formatted string is printed with `putStrLn`. Hence, we can rewrite our previous definitions as follows:

```
showSIF :: String → Int → Float → IO ()
showSIF s i f = 'printf "Name: %s | %+.5i | %+6.3f\n",s,i,f'
```

```
main = showSIF "Curry" 42 3.14159
```

9.1.3 HTML Code

The foreign language tag “html” introduces a notation for HTML expressions (see KiCS2 library HTML) with the standard HTML syntax extended by a layout rule so that closing tags can be omitted. In order to include strings computed by Curry expressions into these HTML syntax, these Curry expressions must be enclosed in curly brackets. The following example program shows its use:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

htmlPage :: String → [HtmlExp]
htmlPage name = ‘‘html
<html>

<head>
  <title>Simple Test

<body>
  <h1>Hello {name}!</h1>
  <p>
    Bye!
  <p>Bye!
  <h2>{reverse name}
  Bye!’’
```

If a Curry expression computes an HTML expression, i.e., it is of type `HtmlExp` instead of `String`, it can be integrated into the HTML syntax by double curly brackets. The following simple example, taken from [14], shows the use of this feature:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

main :: IO HtmlForm
main = return $ form "Question" $
  ‘‘html
    Enter a string: {{textfield tref ""}}
    <hr>
    {{button "Reverse string" revhandler}}
    {{button "Duplicate string" duphandler}}’’

where
  tref free

  revhandler env = return $ form "Answer"
    ‘‘html <h1>Reversed input: {reverse (env tref)}’’
```

```

duphandler env = return $ form "Answer"
  ``html
    <h1>
      Duplicated input:
      {env tref ++ env tref}``

```

9.1.4 XML Expressions

The foreign language tag “xml” introduces a notation for XML expressions (see KiCS2 library XML). The syntax is similar to the language tag “html”, i.e., the use of the layout rule avoids closing tags and Curry expressions evaluating to strings (`String`) and XML expressions (`XmlExp`) can be included by enclosing them in curly and double curly brackets, respectively. The following example program shows its use:

```

{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

import XML

main :: IO ()
main = putStrLn $ showXmlDoc $ head ``xml
  <contact>
    <entry>
      <phone>+49-431-8807271
      <name>Hanus
      <first>Michael
      <email>mh@informatik.uni-kiel.de
      <email>hanus@email.uni-kiel.de

    <entry>
      <name>Smith
      <first>Bill
      <phone>+1-987-742-9388
  ``

```

9.2 Sequential Rules

If the Curry preprocessor is called with the option “seqrules”, then all rules in the Curry module are interpreted in a sequential manner, i.e., a rule is applied only if all previous rules defining the same operation are not applicable, either because the left-hand side’s pattern does not match or the condition is not satisfiable. The idea and detailed semantics of sequential rules are described in [3]. Sequential rules are useful and preferable over rules with multiple guards if the patterns are non-trivial (e.g., functional patterns) or the condition involve complex constraints.

As a simple example, the following module defines a lookup operation in association lists by a functional pattern. Due to the sequential rule strategy, the second rule is applied only if there is no appropriate key in the association list:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=seqrules #-}

mlookup key (_ ++ [(key,value)] ++ _) = Just value
mlookup _ _ = Nothing
```

9.3 Default Rules

An alternative to sequential rules are default rules, i.e., these two options cannot be simultaneously used. Default rules are activated by the preprocessor option “`defaultrules`”. In this case, one can add to each operation a default rule. A default rule for a function f is defined as a rule defining the operation “ f 'default” (this mechanism avoids any language extension for default rules). A default rule is applied only if no “standard” rule is applicable, either because the left-hand sides' pattern do not match or the conditions are not satisfiable.

Default rules are preferable over the sequential rule selection strategy since they have a better operational behavior. This is due to the fact that the test for the application of default rules is done with the same (sometimes optimal) strategy than the selection of standard rules. Moreover, default rules provide a similar power than sequential rules, i.e., they can be applied if the standard rules have complex (functional) patterns or complex conditions.

As a simple example, we show the implementation of the previous example for sequential rules with a default rule:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}

mlookup key (_ ++ [(key,value)] ++ _) = Just value
mlookup'default _ _ = Nothing
```

Default rules are often a good replacement for “negation as failure” used in logic programming. For instance, the following program defines a solution to the n -queens puzzle, where the default rule is useful since it is easier to characterize the unsafe positions of the queens on the chessboard (see the first rule of `safe`):

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}

import Combinatorial(permute)
import Integer(abs)

-- A placement is safe if two queens are not in a same diagonal:
safe (_++[x]++ys++[z]++) | abs (x-z) == length ys + 1 = failed
safe'default xs = xs

-- A solution to the n-queens puzzle is a safe permutation:
queens :: Int → [Int]
queens n = safe (permute [1..n])
```


10 runcurry: Running Curry Programs

`runcurry` is a command usually stored in `kics2home/bin` (where `kics2home` is the installation directory of KiCS2; see Section 1.2). This command supports the execution of Curry programs without explicitly invoking the interactive environment. Hence, it can be useful to write short scripts in Curry intended for direct execution. The Curry program must always contain the definition of an operation `main` of type `I0 ()`. The execution of the program consists of the evaluation of this operation.

Basically, the command `runcurry` supports three modes of operation:

- One can execute a Curry program whose file name is provided as an argument when `runcurry` is called. In this case, the suffix (“`.curry`” or “`.1curry`”) must be present and cannot be dropped. One can write additional commands for the interactive environment, typically settings of some options, before the Curry program name. All arguments after the Curry program name are passed as run-time arguments. For instance, consider the following program stored in the file `ShowArgs.curry`:

```
import System(getArgs)

main = getArgs >>= print
```

This program can be executed by the shell command

```
> runcurry ShowArgs.curry Hello World!
```

which produces the output

```
["Hello","World!"]
```

- One can also execute a Curry program whose whose program text comes from the standard input. Thus, one can either “pipe” the program text into this command or type the program text on the keyboard. For instance, if we type

```
> runcurry
main = putStr . unlines . map show . take 8 $ [1..]
```

(followed by the end-of-file marker `Ctrl-D`), the output

```
1
2
3
4
5
6
7
8
```

is produced.

- One can also write the program text in a script file to be executed like a shell script. In this case, the script must start with the line

```
#!/usr/bin/env runcurry
```

followed by the source text of the Curry program. For instance, we can write a simple Curry script to count the number of code lines in a Curry program by removing all blank and comment lines and counting the remaining lines:

```
#!/usr/bin/env runcurry

import Char(isSpace)
import System(getArgs)

-- count number of program lines in a file:
countCLines :: String -> IO Int
countCLines f =
  readFile f >>=
  return . length . filter (not . isEmptyLine) . map stripSpaces . lines
  where
    stripSpaces = reverse . dropWhile isSpace . reverse . dropWhile isSpace

isEmptyLine []      = True
isEmptyLine [_]     = False
isEmptyLine (c1:c2:_) = c1=='-' && c2=='-'

-- The main program reads Curry file names from arguments:
main = do
  args <- getArgs
  mapIO_ (\f -> do ls <- countCLines f
                  putStrLn $ "Stripped lines of file "++f++": " ++ show ls)
    args
```

If this script is stored in the (executable) file “`codelines.sh`”, we can count the code lines of the file `Prog.curry` by the shell command

```
> ./codelines.sh Prog.curry
```

When this command is executed, the command `runcurry` compiles the program and evaluates the expression `main`. Since the compilation might take some time in more complex scripts, one can also save the result of the compilation in a binary file. To obtain this behavior, one has to insert the line

```
#jit
```

in the script file, e.g., in the second line. With this option, a binary of the compiled program is saved (in the same directory as the script). Now, when the same script is executed the next time, the stored binary file is executed (provided that it is still newer than the script file itself, otherwise it will be recompiled). This feature combines easy scripting with Curry together with fast execution.

11 ERD2Curry: A Tool to Generate Programs from ER Specifications

ERD2Curry is a tool to generate Curry code to access and manipulate data persistently stored from entity relationship diagrams. The idea of this tool is described in detail in [8]. Thus, we describe only the basic steps to use this tool in the following.

If one creates an entity relationship diagram (ERD) with the Umbrello UML Modeller, one has to store its XML description in XMI format (as offered by Umbrello) in a file, e.g., “myerd.xmi”. This description can be compiled into a Curry program by the command

```
erd2curry -x myerd.xmi
```

(`erd2curry` is a program stored in `kics2home/bin` where `kics2home` is the installation directory of KiCS2; see Section 1.2). If `MyData` is the name of the ERD, the Curry program file “`MyData.curry`” is generated containing all the necessary database access code as described in [8]. In addition to the generated Curry program file, two auxiliary program files `ERDGeneric.curry` and `KeyDatabase.curry` are created in the same directory.

If one does not want to use the Umbrello UML Modeller, which might be the preferred method since the interface to the Umbrello UML Modeller is no longer actively supported, one can also create a textual description of the ERD as a Curry term of type `ERD` (w.r.t. the type definition given in module `kics2home/currytools/erd2curry/ERD.curry`) and store it in some file, e.g., “myerd.term”. This description can be compiled into a Curry program by the command

```
erd2curry -t myerd.term
```

The directory `kics2home/currytools/erd2curry/` contains two examples for such ERD term files:

Blog.erdterm: This is a simple ERD model for a blog with entries, comments, and tags.

Uni.erdterm: This is an ERD model for university lectures as presented in the paper [8].

There is also the possibility to visualize an ERD term as a graph with the graph visualization program `dotty` (for this purpose, it might be necessary to adapt the definition of `dotviewcommand` in your “.kics2rc” file, see Section 2.7, according to your local environment). The visualization can be performed by the command

```
erd2curry -v myerd.term
```

12 Spicely: An ER-based Web Framework

Spicely is a framework to support the implementation of web-based systems in Curry. Spicely generates an initial implementation from an entity-relationship (ER) description of the underlying data. The generated implementation contains operations to create and manipulate entities of the data model, supports authentication, authorization, session handling, and the composition of individual operations to user processes. Furthermore, the implementation ensures the consistency of the database w.r.t. the data dependencies specified in the ER model, i.e., updates initiated by the user cannot lead to an inconsistent state of the database.

The idea of this tool, which is part of the distribution of KiCS2, is described in detail in [18]. Thus, we describe only the basic steps to use this tool in order to generate a web application.

First, one has to create a textual description of the entity-relationship model as a Curry term of type ERD (w.r.t. the type definitions given in module `kics2home/currytools/erd2curry/ERD.curry`) and store it in some file, e.g., “`mymodel.erdterm`”. The directory `kics2home/currytools/spicely/` contains two examples for such ERD term files:

Blog.erdterm: This is a simple ER model for a blog with entries, comments, and tags, as presented in the paper [18].

Uni.erdterm: This is an ER model for university lectures as presented in the paper [8].

Then change to the directory in which you want to create the project sources. Execute the command

```
spiceup ../mymodel.erdterm
```

with the path to the ERD term file as a parameter (`spiceup` is a program stored in `kics2home/bin` where `kics2home` is the installation directory of KiCS2; see Section 1.2). You can also provide a path name, i.e., the name of a directory, where the database files should be stored, e.g.,

```
spiceup --dbpath DBDIR ../mymodel.erdterm
```

If the parameter “`--dbpath DBDIR`” is not provided, then `DBDIR` is set to the current directory (“.”). Since this specification will be used in the *generated* web programs, a relative database directory name will be relative to the place where the web programs are stored. In order to avoid such confusion, it might be better to specify an absolute path name for the database directory.

After the generation of this project (see the generated file `README.txt` for information about the generated project structure), one can compile the generated programs by

```
make compile
```

In order to generate the executable web application, configure the generated `Makefile` by adapting the variable `WEBSERVERDIR` to the location where the compiled cgi programs should be stored, and run

```
make deploy
```

After the successful compilation and deployment of all files, the application is executable in a web browser by selecting the URL `<URL of web dir>/spicely.cgi`.

13 Technical Problems

One can implement distributed systems with KiCS2 by the use of the library `NamedSocket` (Section A.2.28) that supports a socket communication with symbolic names rather than natural numbers. For instance, this library is the basis of programming dynamic web pages with the libraries `HTML` (Section A.4.2) or `WUI` (Section A.4.8). However, it might be possible that some technical problems arise due to the use of named sockets. Therefore, this section gives some information about the technical requirements of KiCS2 and how to solve problems due to these requirements.

There is one fixed port that is used by the implementation of KiCS2:

Port 8767: This port is used by the **Curry Port Name Server** (CPNS) to implement symbolic names for named sockets in Curry. If some other process uses this port on the machine, the distribution facilities defined in the module `NamedSocket` cannot be used.

If these features do not work, you can try to find out whether this port is in use by the shell command “`netstat -a | fgrep 8767`” (or similar).

The CPNS is implemented as a demon listening on its port 8767 in order to serve requests about registering a new symbolic name for a named socket or asking the physical port number of an registered named socket. The demon will be automatically started for the first time on a machine when a user runs a program using named sockets. It can also be manually started and terminated by the scripts `kics2home/cpns/start` and `kics2home/cpns/stop`. If the demon is already running, the command `kics2home/cpns/start` does nothing (so it can be always executed before invoking a Curry program using named sockets).

If you detect any further technical problem, please write to

`kics2@curry-language.org`

References

- [1] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
- [2] S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
- [3] S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.
- [4] S. Antoy and M. Hanus. From boolean equalities to constraints. In *Proceedings of the 25th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2015)*. Springer LNCS (to appear), 2015.
- [5] S. Antoy and M. Hanus. Default rules for Curry. In *Proc. of the 18th International Symposium on Practical Aspects of Declarative Languages (PADL 2016)*. Springer LNCS (to appear), 2016.
- [6] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.
- [7] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
- [8] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.
- [9] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011)*, pages 22–33. INFSYS Research Report 1843-11-06 (TU Wien), 2011.
- [10] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
- [11] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
- [12] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

- [13] M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.
- [14] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
- [15] M. Hanus. A generic analysis environment for declarative programs. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 43–48. ACM Press, 2005.
- [16] M. Hanus. CurryBrowser: A generic analysis environment for Curry programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pages 61–74, 2006.
- [17] M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
- [18] M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. *Theory and Practice of Logic Programming*, 14(3):269–291, 2014.
- [19] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.
- [20] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at <http://www.curry-language.org>, 2015.
- [21] T. Johnsson. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer LNCS 201, 1985.
- [22] P. Wadler. Efficient compilation of pattern-matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

A Libraries of the KiCS2 Distribution

The KiCS2 distribution comes with an extensive collection of libraries for application programming. The libraries for meta-programming by representing Curry programs as datatypes in Curry are described in the following subsection in more detail. The complete set of libraries with all exported types and functions are described in the further subsections. For a more detailed online documentation of all libraries of KiCS2, see <http://www-ps.informatik.uni-kiel.de/kics2/lib/index.html>.

A.1 AbstractCurry and FlatCurry: Meta-Programming in Curry

To support meta-programming, i.e., the manipulation of Curry programs in Curry, there are system modules `FlatCurry.Types` (Section A.5.6) and `AbstractCurry.Types` (Section A.5.1) which define datatypes for the representation of Curry programs. `AbstractCurry.Types` is a more direct representation of a Curry program, whereas `FlatCurry.Types` is a simplified representation where local function definitions are replaced by global definitions (i.e., lambda lifting has been performed) and pattern matching is translated into explicit case/or expressions. Thus, `FlatCurry.Types` can be used for more back-end oriented program manipulations (or, for writing new back ends for Curry), whereas `AbstractCurry.Types` is intended for manipulations of programs that are more oriented towards the source program.

There are predefined I/O actions to read `AbstractCurry` and `FlatCurry` programs: `AbstractCurry.Files.readCurry` and `FlatCurry.Files.readFlatCurry`. These actions parse the corresponding source program and return a data term representing this program (according to the definitions in the modules `AbstractCurry.Types` and `FlatCurry.Types`).

Since all datatypes are explained in detail in these modules, we refer to the online documentation⁵ of these modules.

As an example, consider a program file “`test.curry`” containing the following two lines:

```
rev []      = []
rev (x:xs) = (rev xs) ++ [x]
```

Then the I/O action (`FlatCurry.Files.readFlatCurry "test"`) returns the following term:

```
(Prog "test"
  ["Prelude"]
  []
  [Func ("test","rev") 1 Public
    (FuncType (TCons ("Prelude","[]") [(TVar 0)])
              (TCons ("Prelude","[]") [(TVar 0)]))
    (Rule [0]
      (Case Flex (Var 1)
        [Branch (Pattern ("Prelude","[]") [])
          (Comb ConsCall ("Prelude","[]") []),
          Branch (Pattern ("Prelude",":") [2,3])
            (Comb FuncCall ("Prelude", "++")
              [Comb FuncCall ("test","rev") [Var 3],
```

⁵<http://www-ps.informatik.uni-kiel.de/kics2/lib/FlatCurry.Types.html> and <http://www-ps.informatik.uni-kiel.de/kics2/lib/AbstractCurry.Types.html>


```

        Comb ConsCall ("Prelude",":")
          [Var 2,Comb ConsCall ("Prelude","[]") []]
      ])
    ]))
  []
)

```

A.2 General Libraries

A.2.1 Library AllSolutions

This module contains a collection of functions for obtaining lists of solutions to constraints. These operations are useful to encapsulate non-deterministic operations between I/O actions in order to connect the worlds of logic and functional programming and to avoid non-determinism failures on the I/O level.

In contrast the "old" concept of encapsulated search (which could be applied to any subexpression in a computation), the operations to encapsulate search in this module are I/O actions in order to avoid some anomalies in the old concept.

Exported functions:

`getAllValues :: a → IO [a]`

Gets all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

`getOneValue :: a → IO (Maybe a)`

Gets one value of an expression (currently, via an incomplete left-to-right strategy). Returns `Nothing` if the search space is finitely failed.

`getAllSolutions :: (a → Bool) → IO [a]`

Gets all solutions to a constraint (currently, via an incomplete depth-first left-to-right strategy). Conceptually, all solutions are computed on a copy of the constraint, i.e., the evaluation of the constraint does not share any results. Moreover, this evaluation suspends if the constraints contain unbound variables. Similar to Prolog's `findall`.

`getOneSolution :: (a → Bool) → IO (Maybe a)`

Gets one solution to a constraint (currently, via an incomplete left-to-right strategy). Returns `Nothing` if the search space is finitely failed.

`getAllFailures :: a → (a → Bool) → IO [a]`

Returns a list of values that do not satisfy a given constraint.

A.2.2 Library Assertion

This module defines the datatype and operations for the Curry module tester "currytest".

Exported types:

`data Assertion`

Datatype for defining test cases.

Exported constructors:

`data ProtocolMsg`

The messages sent to the test GUI. Used by the currytest tool.

Exported constructors:

- `TestModule :: String → ProtocolMsg`
- `TestCase :: String → Bool → ProtocolMsg`
- `TestFinished :: ProtocolMsg`
- `TestCompileError :: ProtocolMsg`

Exported functions:

`assertTrue :: String → Bool → Assertion ()`

`(assertTrue s b)` asserts (with name `s`) that `b` must be true.

`assertEqual :: String → a → a → Assertion a`

`(assertEqual s e1 e2)` asserts (with name `s`) that `e1` and `e2` must be equal (w.r.t. `==`).

`assertValues :: String → a → [a] → Assertion a`

`(assertValues s e vs)` asserts (with name `s`) that `vs` is the multiset of all values of `e`. All values of `e` are compared with the elements in `vs` w.r.t. `==`.

`assertSolutions :: String → (a → Bool) → [a] → Assertion a`

`(assertSolutions s c vs)` asserts (with name `s`) that constraint abstraction `c` has the multiset of solutions `vs`. The solutions of `c` are compared with the elements in `vs` w.r.t. `==`.

`assertIO :: String → IO a → a → Assertion a`

`(assertIO s a r)` asserts (with name `s`) that I/O action `a` yields the result value `r`.

`assertEqualIO :: String → IO a → IO a → Assertion a`

`(assertEqualIO s a1 a2)` asserts (with name `s`) that I/O actions `a1` and `a2` yield equal (w.r.t. `==`) results.

`seqStrActions :: IO (String,Bool) → IO (String,Bool) → IO (String,Bool)`

Combines two actions and combines their results. Used by the `currytest` tool.

`checkAssertion :: String → ((String,Bool) → IO (String,Bool)) → Assertion a → IO (String,Bool)`

Executes and checks an assertion, and process the result by an I/O action. Used by the `currytest` tool.

`writeAssertResult :: (String,Bool) → IO Int`

Prints the results of assertion checking. If failures occurred, the return code is positive. Used by the `currytest` tool.

`showTestMod :: Int → String → IO ()`

Sends message to GUI for showing test of a module. Used by the `currytest` tool.

`showTestCase :: Int → (String,Bool) → IO (String,Bool)`

Sends message to GUI for showing result of executing a test case. Used by the `currytest` tool.

`showTestEnd :: Int → IO ()`

Sends message to GUI for showing end of module test. Used by the `currytest` tool.

`showTestCompileError :: Int → IO ()`

Sends message to GUI for showing compilation errors in a module test. Used by the `currytest` tool.

A.2.3 Library Char

Library with some useful functions on characters.

Exported functions:

`isAscii :: Char → Bool`

Returns true if the argument is an ASCII character.

`isLatin1 :: Char → Bool`

Returns true if the argument is an Latin-1 character.

`isAsciiLower :: Char → Bool`

Returns true if the argument is an ASCII lowercase letter.

`isAsciiUpper :: Char → Bool`

Returns true if the argument is an ASCII uppercase letter.

`isControl :: Char → Bool`

Returns true if the argument is a control character.

`isUpper :: Char → Bool`

Returns true if the argument is an uppercase letter.

`isLower :: Char → Bool`

Returns true if the argument is an lowercase letter.

`isAlpha :: Char → Bool`

Returns true if the argument is a letter.

`isDigit :: Char → Bool`

Returns true if the argument is a decimal digit.

`isAlphaNum :: Char → Bool`

Returns true if the argument is a letter or digit.

`isBinDigit :: Char → Bool`

Returns true if the argument is a binary digit.

`isOctDigit :: Char → Bool`

Returns true if the argument is an octal digit.

`isHexDigit :: Char → Bool`

Returns true if the argument is a hexadecimal digit.

`isSpace :: Char → Bool`

Returns true if the argument is a white space.

`toUpper :: Char → Char`

Converts lowercase into uppercase letters.

`toLower :: Char → Char`

Converts uppercase into lowercase letters.

`digitToInt :: Char → Int`

Converts a (hexadecimal) digit character into an integer.

`intToDigit :: Int → Char`

Converts an integer into a (hexadecimal) digit character.

A.2.4 Library Combinatorial

A collection of common non-deterministic and/or combinatorial operations. Many operations are intended to operate on sets. The representation of these sets is not hidden; rather sets are represented as lists. Ideally these lists contains no duplicate elements and the order of their elements cannot be observed. In practice, these conditions are not enforced.

Exported functions:

`permute :: [a] → [a]`

Compute any permutation of a list. For example, `[1,2,3,4]` may give `[1,3,4,2]`.

`subset :: [a] → [a]`

Compute any sublist of a list. The sublist contains some of the elements of the list in the same order. For example, `[1,2,3,4]` may give `[1,3]`, and `[1,2,3]` gives `[1,2,3]`, `[1,2]`, `[1,3]`, `[1]`, `[2,3]`, `[2]`, `[3]`, or `[]`.

`splitSet :: [a] → ([a], [a])`

Split a list into any two sublists. For example, `[1,2,3,4]` may give `([1,3,4],[2])`.

`sizedSubset :: Int → [a] → [a]`

Compute any sublist of fixed length of a list. Similar to `subset`, but the length of the result is fixed.

`partition :: [a] → [[a]]`

Compute any partition of a list. The output is a list of non-empty lists such that their concatenation is a permutation of the input list. No guarantee is made on the order of the arguments in the output. For example, `[1,2,3,4]` may give `[[4],[2,3],[1]]`, and `[1,2,3]` gives `[[1,2,3]]`, `[[2,3],[1]]`, `[[1,3],[2]]`, `[[3],[1,2]]`, or `[[3],[2],[1]]`.

A.2.5 Library Constraint

Some useful operations for constraint programming.

Exported functions:

`(<:) :: a → a → Bool`

Less-than on ground data terms as a constraint.

`(>:) :: a → a → Bool`

Greater-than on ground data terms as a constraint.

`(<=:) :: a → a → Bool`

Less-or-equal on ground data terms as a constraint.

`(>=:) :: a → a → Bool`

Greater-or-equal on ground data terms as a constraint.

`andC :: [Bool] → Bool`

Evaluates the conjunction of a list of constraints.

`orC :: [Bool] → Bool`

Evaluates the disjunction of a list of constraints.

`allC :: (a → Bool) → [a] → Bool`

Is a given constraint abstraction satisfied by all elements in a list?

`anyC :: (a → Bool) → [a] → Bool`

Is there an element in a list satisfying a given constraint?

A.2.6 Library CPNS

Implementation of a Curry Port Name Server based on raw sockets. It is used to implement the library Ports for distributed programming with ports.

Exported functions:

`cpnsStart :: IO ()`

Starts the "Curry Port Name Server" (CPNS) running on the local machine. The CPNS is responsible to resolve symbolic names for ports into physical socket numbers so that a port can be reached under its symbolic name from any machine in the world.

`cpnsShow :: IO ()`

Shows all registered ports at the local CPNS demon (in its logfile).

`cpnsStop :: IO ()`

Terminates the local CPNS demon

`registerPort :: String → Int → Int → IO ()`

Registers a symbolic port at the local host.

`getPortInfo :: String → String → IO (Int,Int)`

Gets the information about a symbolic port at some host.

`unregisterPort :: String → IO ()`

Unregisters a symbolic port at the local host.

`cpnsAlive :: Int → String → IO Bool`

Tests whether the CPNS demon at a host is alive.

`main :: IO ()`

Main function for CPNS demon. Check arguments and execute command.

A.2.7 Library CSV

Library for reading/writing files in CSV format. Files in CSV (comma separated values) format can be imported and exported by most spreadsheed and database applications.

Exported functions:

`writeCSVFile :: String → [[String]] → IO ()`

Writes a list of records (where each record is a list of strings) into a file in CSV format.

`showCSV :: [[String]] → String`

Shows a list of records (where each record is a list of strings) as a string in CSV format.

`readCSVFile :: String → IO [[String]]`

Reads a file in CSV format and returns the list of records (where each record is a list of strings).

`readCSVFileWithDelims :: String → String → IO [[String]]`

Reads a file in CSV format and returns the list of records (where each record is a list of strings).

`readCSV :: String → [[String]]`

Reads a string in CSV format and returns the list of records (where each record is a list of strings).

`readCSVWithDelims :: String → String → [[String]]`

Reads a string in CSV format and returns the list of records (where each record is a list of strings).

A.2.8 Library Debug

This library contains some useful operation for debugging programs.

Exported functions:

`trace :: String → a → a`

Prints the first argument as a side effect and behaves as identity on the second argument.

`traceId :: String → String`

Prints the first argument as a side effect and returns it afterwards.

`traceShow :: a → b → b`

Prints the first argument using `show` and returns the second argument afterwards.

`traceShowId :: a → a`

Prints the first argument using `show` and returns it afterwards.

`traceIO :: String → IO ()`

Output a trace message from the IO monad.

`assert :: Bool → String → a → a`

Assert a condition w.r.t. an error message. If the condition is not met it fails with the given error message, otherwise the third argument is returned.

`assertIO :: Bool → String → IO ()`

Assert a condition w.r.t. an error message from the IO monad. If the condition is not met it fails with the given error message.

A.2.9 Library Directory

Library for accessing the directory structure of the underlying operating system.

Exported functions:

`doesFileExist :: String → IO Bool`

Returns true if the argument is the name of an existing file.

`doesDirectoryExist :: String → IO Bool`

Returns true if the argument is the name of an existing directory.

`fileSize :: String → IO Int`

Returns the size of the file.

`getModificationTime :: String → IO ClockTime`

Returns the modification time of the file.

`getCurrentDirectory :: IO String`

Returns the current working directory.

`setCurrentDirectory :: String → IO ()`

Sets the current working directory.

`getDirectoryContents :: String → IO [String]`

Returns the list of all entries in a directory.

`createDirectory :: String → IO ()`

Creates a new directory with the given name.

`createDirectoryIfMissing :: Bool → String → IO ()`

Creates a new directory with the given name if it does not already exist. If the first parameter is `True` it will also create all missing parent directories.

`removeDirectory :: String → IO ()`

Deletes a directory from the file system.

`renameDirectory :: String → String → IO ()`

Renames a directory.

`getHomeDirectory :: IO String`

Returns the home directory of the current user.

`getTemporaryDirectory :: IO String`

Returns the temporary directory of the operating system.

`getAbsolutePath :: String → IO String`

Convert a path name into an absolute one. For instance, a leading `~` is replaced by the current home directory.

`removeFile :: String → IO ()`

Deletes a file from the file system.

`renameFile :: String → String → IO ()`

Renames a file.

`copyFile :: String → String → IO ()`

Copy the contents from one file to another file

A.2.10 Library Distribution

This module contains functions to obtain information concerning the current distribution of the Curry implementation, e.g., compiler version, load paths, front end.

Exported types:

`data FrontendTarget`

Data type for representing the different target files that can be produced by the front end of the Curry compiler.

Exported constructors:

- `FCY :: FrontendTarget`
FCY

- FlatCurry file ending with .fcy
- `FINT :: FrontendTarget`
`FINT`
 - FlatCurry interface file ending with .fint
- `ACY :: FrontendTarget`
`ACY`
 - AbstractCurry file ending with .acy
- `UACY :: FrontendTarget`
`UACY`
 - Untyped (without type checking) AbstractCurry file ending with .uacy
- `HTML :: FrontendTarget`
`HTML`
 - colored HTML representation of source program
- `CY :: FrontendTarget`
`CY`
 - source representation employed by the frontend

`data FrontendParams`

Abstract data type for representing parameters supported by the front end of the Curry compiler.

Exported constructors:

Exported functions:

`curryCompiler :: String`

The name of the Curry compiler (e.g., "pakcs" or "kics2").

`curryCompilerMajorVersion :: Int`

The major version number of the Curry compiler.

`curryCompilerMinorVersion :: Int`

The minor version number of the Curry compiler.

`curryRuntime :: String`

The name of the run-time environment (e.g., "sicstus", "swi", or "ghc")

`curryRuntimeMajorVersion :: Int`

The major version number of the Curry run-time environment.

`curryRuntimeMinorVersion :: Int`

The minor version number of the Curry run-time environment.

`installDir :: String`

Path of the main installation directory of the Curry compiler.

`rcFileName :: IO String`

The name of the file specifying configuration parameters of the current distribution. This file must have the usual format of property files (see description in module `PropertyFile`).

`rcFileContents :: IO [(String,String)]`

Returns the current configuration parameters of the distribution. This action yields the list of pairs (var,val).

`getRcVar :: String → IO (Maybe String)`

Look up a specific configuration variable as specified by user in his rc file. Uppercase/lowercase is ignored for the variable names.

`getRcVars :: [String] → IO [Maybe String]`

Look up configuration variables as specified by user in his rc file. Uppercase/lowercase is ignored for the variable names.

`splitModuleFileName :: String → String → (String,String)`

Split the `FilePath` of a module into the directory prefix and the `FilePath` corresponding to the module name. For instance, the call `splitModuleFileName "Data.Set" "lib/Data/Set.curry"` evaluates to `("lib", "Data/Set.curry")`. This can be useful to compute output directories while retaining the hierarchical module structure.

`splitModuleIdentifiers :: String → [String]`

Split up the components of a module identifier. For instance, `splitModuleIdentifiers "Data.Set"` evaluates to `["Data", "Set"]`.

`joinModuleIdentifiers :: [String] → String`

Join the components of a module identifier. For instance, `joinModuleIdentifiers ["Data", "Set"]` evaluates to `"Data.Set"`.

`stripCurrySuffix :: String → String`

Strips the suffix `".curry"` or `".lcurry"` from a file name.

`moduleNameToPath :: String → String`

Transforms a hierarchical module name into a path name, i.e., replace the dots in the name by directory separator chars.

`currySubdir :: String`

Name of the sub directory where auxiliary files (.fint, .fcy, etc) are stored.

`inCurrySubdir :: String → String`

Transforms a path to a module name into a file name by adding the `currySubDir` to the path and transforming a hierarchical module name into a path. For instance, `inCurrySubdir "mylib/Data.Char"` evaluates to `"mylib/.curry/Data/Char"`.

`inCurrySubdirModule :: String → String → String`

Transforms a file name by adding the `currySubDir` to the file name. This version respects hierarchical module names.

`addCurrySubdir :: String → String`

Transforms a directory name into the name of the corresponding sub directory containing auxiliary files.

`getLoadPathForModule :: String → IO [String]`

Returns the current path (list of directory names) that is used for loading modules w.r.t. a given module path. The directory prefix of the module path (or "." if there is no such prefix) is the first element of the load path and the remaining elements are determined by the environment variable `CURRYRPATH` and the entry "libraries" of the system's rc file.

`lookupModuleSourceInLoadPath :: String → IO (Maybe (String,String))`

Returns a directory name and the actual source file name for a module by looking up the module source in the current load path. If the module is hierarchical, the directory is the top directory of the hierarchy. Returns `Nothing` if there is no corresponding source file.

`defaultParams :: FrontendParams`

The default parameters of the front end.

`rcParams :: IO FrontendParams`

The default parameters of the front end as configured by the compiler specific resource configuration file.

`setQuiet :: Bool → FrontendParams → FrontendParams`

Set quiet mode of the front end.

`setExtended :: Bool → FrontendParams → FrontendParams`

Set extended mode of the front end.

`setOverlapWarn :: Bool → FrontendParams → FrontendParams`

Set overlap warn mode of the front end.

`setFullPath :: [String] → FrontendParams → FrontendParams`

Set the full path of the front end. If this parameter is set, the front end searches all modules in this path (instead of using the default path).

`setHtmlDir :: String → FrontendParams → FrontendParams`

Set the `htmldir` parameter of the front end. Relevant for HTML generation.

`setLogfile :: String → FrontendParams → FrontendParams`

Set the `logfile` parameter of the front end. If this parameter is set, all messages produced by the front end are stored in this file.

`setSpecials :: String → FrontendParams → FrontendParams`

Set additional specials parameters of the front end. These parameters are specific for the current front end and should be used with care, since their form might change in the future.

`quiet :: FrontendParams → Bool`

Returns the value of the "quiet" parameter.

`extended :: FrontendParams → Bool`

Returns the value of the "extended" parameter.

`overlapWarn :: FrontendParams → Bool`

Returns the value of the "overlapWarn" parameter.

`fullPath :: FrontendParams → Maybe [String]`

Returns the full path parameter of the front end.

`htmldir :: FrontendParams → Maybe String`

Returns the `htmldir` parameter of the front end.

`logfile :: FrontendParams → Maybe String`

Returns the `logfile` parameter of the front end.

`specials :: FrontendParams → String`

Returns the special parameters of the front end.

`callFrontend :: FrontendTarget → String → IO ()`

In order to make sure that compiler generated files (like `.fcy`, `.fint`, `.acy`) are up to date, one can call the front end of the Curry compiler with this action. If the front end returns with an error, an exception is raised.

`callFrontendWithParams :: FrontendTarget → FrontendParams → String → IO ()`

In order to make sure that compiler generated files (like `.fcy`, `.fint`, `.acy`) are up to date, one can call the front end of the Curry compiler with this action where various parameters can be set. If the front end returns with an error, an exception is raised.

A.2.11 Library `Either`

Library with some useful operations for the `Either` data type.

Exported functions:

`lefts :: [Either a b] → [a]`

Extracts from a list of `Either` all the `Left` elements in order.

`rights :: [Either a b] → [b]`

Extracts from a list of `Either` all the `Right` elements in order.

`isLeft :: Either a b → Bool`

Return `True` if the given value is a `Left`-value, `False` otherwise.

`isRight :: Either a b → Bool`

Return `True` if the given value is a `Right`-value, `False` otherwise.

`fromLeft :: Either a b → a`

Extract the value from a `Left` constructor.

`fromRight :: Either a b → b`

Extract the value from a `Right` constructor.

`partitionEithers :: [Either a b] → ([a],[b])`

Partitions a list of `Either` into two lists. All the `Left` elements are extracted, in order, to the first component of the output. Similarly the `Right` elements are extracted to the second component of the output.

A.2.12 Library `ErrorState`

A combination of `Error` and state monad like `ErrorT State` in Haskell.

Exported types:

`type ES a b c = b → Either a (c,b)`

Error state monad.

Exported functions:

`evalES :: (a → Either b (c,a)) → a → Either b c`

Evaluate an ES monad

`returnES :: a → b → Either c (a,b)`

Lift a value into the ES monad

`failES :: a → b → Either a (c,b)`

Failing computation in the ES monad

`(>+)= :: (a → Either b (c,a)) → (c → a → Either b (d,a)) → a → Either b (d,a)`

Bind of the ES monad

`(>+) :: (a → Either b (c,a)) → (a → Either b (d,a)) → a → Either b (d,a)`

Sequence operator of the ES monad

`(<$>) :: (a → b) → (c → Either d (a,c)) → c → Either d (b,c)`

Apply a pure function onto a monadic value.

`(<*>) :: (a → Either b (c → d,a)) → (a → Either b (c,a)) → a → Either b (d,a)`

Apply a function yielded by a monadic action to a monadic value.

`gets :: a → Either b (a,a)`

Retrieve the current state

`puts :: a → a → Either b ((),a)`

Replace the current state

`modify :: (a → a) → a → Either b ((),a)`

Modify the current state

`mapES :: (a → b → Either c (d,b)) → [a] → b → Either c ([d],b)`

Map a monadic function on all elements of a list by sequencing the effects.

`concatMapES :: (a → b → Either c ([d],b)) → [a] → b → Either c ([d],b)`

Same as `concatMap`, but for a monadic function.

`mapAccumES :: (a → b → c → Either d ((a,e),c)) → a → [b] → c → Either d ((a,[e]),c)`

Same as `mapES` but with an additional accumulator threaded through.

A.2.13 Library FileGoodies

A collection of useful operations when dealing with files.

Exported functions:

`separatorChar :: Char`

The character for separating hierarchies in file names. On UNIX systems the value is `/`.

`pathSeparatorChar :: Char`

The character for separating names in path expressions. On UNIX systems the value is `..`.

`suffixSeparatorChar :: Char`

The character for separating suffixes in file names. On UNIX systems the value is `..`.

`isAbsolute :: String → Bool`

Is the argument an absolute name?

`dirName :: String → String`

Extracts the directory prefix of a given (Unix) file name. Returns `..` if there is no prefix.

`baseName :: String → String`

Extracts the base name without directory prefix of a given (Unix) file name.

`splitDirectoryBaseName :: String → (String,String)`

Splits a (Unix) file name into the directory prefix and the base name. The directory prefix is `..` if there is no real prefix in the name.

`stripSuffix :: String → String`

Strips a suffix (the last suffix starting with a dot) from a file name.

`fileSuffix :: String → String`

Yields the suffix (the last suffix starting with a dot) from given file name.

`splitBaseName :: String → (String,String)`

Splits a file name into prefix and suffix (the last suffix starting with a dot and the rest).

`splitPath :: String → [String]`

Splits a path string into list of directory names.

`lookupFileInPath :: String → [String] → [String] → IO (Maybe String)`

Looks up the first file with a possible suffix in a list of directories. Returns `Nothing` if such a file does not exist.

`getFileInPath :: String → [String] → [String] → IO String`

Gets the first file with a possible suffix in a list of directories. An error message is delivered if there is no such file.

A.2.14 Library FilePath

This library is a direct port of the Haskell library `System.FilePath` of Neil Mitchell.

Exported types:

```
type FilePath = String
```

Exported functions:

```
pathSeparator :: Char
```

```
pathSeparators :: String
```

```
isPathSeparator :: Char → Bool
```

```
searchPathSeparator :: Char
```

```
isSearchPathSeparator :: Char → Bool
```

```
extSeparator :: Char
```

```
isExtSeparator :: Char → Bool
```

```
splitSearchPath :: String → [String]
```

```
getSearchPath :: IO [String]
```

```
splitExtension :: String → (String, String)
```

```
takeExtension :: String → String
```

replaceExtension :: String → String → String

(<.>) :: String → String → String

dropExtension :: String → String

addExtension :: String → String → String

hasExtension :: String → Bool

splitExtensions :: String → (String,String)

dropExtensions :: String → String

takeExtensions :: String → String

splitDrive :: String → (String,String)

joinDrive :: String → String → String

takeDrive :: String → String

dropDrive :: String → String

hasDrive :: String → Bool

isDrive :: String → Bool

```
splitFileName :: String → (String,String)

replaceFileName :: String → String → String

dropFileName :: String → String

takeFileName :: String → String

takeBaseName :: String → String

replaceBaseName :: String → String → String

hasTrailingPathSeparator :: String → Bool

addTrailingPathSeparator :: String → String

dropTrailingPathSeparator :: String → String

takeDirectory :: String → String

replaceDirectory :: String → String → String

combine :: String → String → String

(</>) :: String → String → String

splitPath :: String → [String]
```

```
splitDirectories :: String → [String]
```

```
joinPath :: [String] → String
```

```
equalFilePath :: String → String → Bool
```

```
makeRelative :: String → String → String
```

```
normalise :: String → String
```

```
isValid :: String → Bool
```

```
makeValid :: String → String
```

```
isRelative :: String → Bool
```

```
isAbsolute :: String → Bool
```

A.2.15 Library Findall

Library with some operations for encapsulating search. Note that some of these operations are not fully declarative, i.e., the results depend on the order of evaluation and program rules. There are newer and better approaches the encapsulate search, in particular, set functions (see module `SetFunctions`), which should be used.

This library is included only for compatibility with PAKCS.

Exported functions:

```
getAllValues :: a → IO [a]
```

Gets all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables. Similar to Prolog's `findall`.

`getSomeValue :: a → IO a`

Gets a value of an expression (currently, via an incomplete depth-first strategy). The expression must have a value, otherwise the computation fails. Conceptually, the value is computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

`allValues :: a → [a]`

Returns all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules.

`someValue :: a → a`

Returns some value for an expression (currently, via an incomplete depth-first strategy). If the expression has no value, the computation fails. Conceptually, the value is computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value.

`allSolutions :: (a → Bool) → [a]`

Returns all values satisfying a predicate, i.e., all arguments such that the predicate applied to the argument can be evaluated to `True` (currently, via an incomplete depth-first strategy).

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules.

`someSolution :: (a → Bool) → a`

Returns some values satisfying a predicate, i.e., some argument such that the predicate applied to the argument can be evaluated to `True` (currently, via an incomplete depth-first strategy). If there is no value satisfying the predicate, the computation fails.

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules. Thus, this operation should be used only if the predicate has a single solution.

A.2.16 Library Float

A collection of operations on floating point numbers.

Exported functions:

`pi :: Float`

The number pi.

`(+.) :: Float → Float → Float`

Addition on floats.

`(-.) :: Float → Float → Float`

Subtraction on floats.

`(*.) :: Float → Float → Float`

Multiplication on floats.

`(/.) :: Float → Float → Float`

Division on floats.

`(^.) :: Float → Int → Float`

The value of `a ^. b` is `a` raised to the power of `b`. Executes in $O(\log b)$ steps.

`i2f :: Int → Float`

Conversion function from integers to floats.

`truncate :: Float → Int`

Conversion function from floats to integers. The result is the closest integer between the argument and 0.

`round :: Float → Int`

Conversion function from floats to integers. The result is the nearest integer to the argument. If the argument is equidistant between two integers, it is rounded to the closest even integer value.

`recip :: Float → Float`

Reciprocal

`sqrt :: Float → Float`

Square root.

`log :: Float → Float`

Natural logarithm.

`logBase :: Float → Float → Float`

Logarithm to arbitrary Base.

`exp :: Float → Float`

Natural exponent.

`sin :: Float → Float`

Sine.

`cos :: Float → Float`

Cosine.

`tan :: Float → Float`

Tangent.

`asin :: Float → Float`

Arc sine.

`acos :: Float → Float`

`atan :: Float → Float`

Arc tangent.

`sinh :: Float → Float`

Hyperbolic sine.

`cosh :: Float → Float`

`tanh :: Float → Float`

Hyperbolic tangent.

`asinh :: Float → Float`

Hyperbolic Arc sine.

`acosh :: Float → Float`

`atanh :: Float → Float`

Hyperbolic Arc tangent.

A.2.17 Library Function

This module provides some utility functions for function application.

Exported functions:

`fix :: (a → a) → a`

`fix f` is the least fixed point of the function `f`, i.e. the least defined `x` such that `f x = x`.

`on :: (a → a → b) → (c → a) → c → c → b`

(*) ‘on’ `f = \x y -> f x * f y`. Typical usage: `sortBy (compare ‘on’ fst)`.

`first :: (a → b) → (a,c) → (b,c)`

Apply a function to the first component of a tuple.

`second :: (a → b) → (c,a) → (c,b)`

Apply a function to the second component of a tuple.

(***) `:: (a → b) → (c → d) → (a,c) → (b,d)`

Apply two functions to the two components of a tuple.

(&&&) `:: (a → b) → (a → c) → a → (b,c)`

Apply two functions to a value and returns a tuple of the results.

`both :: (a → b) → (a,a) → (b,b)`

Apply a function to both components of a tuple.

A.2.18 Library `GetOpt`

This Module is a modified version of the Module `System.Console.GetOpt` by Sven Panne from the `ghc-base` package it has been adapted for Curry by Bjoern Peemoeller

(c) Sven Panne 2002-2005 The Glasgow Haskell Compiler License

Copyright 2004, The University Court of the University of Glasgow. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

this list of conditions and the following disclaimer.

this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED

TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Exported types:

data ArgOrder

Exported constructors:

- RequireOrder :: ArgOrder a
- Permute :: ArgOrder a
- ReturnInOrder :: (String → a) → ArgOrder a

data OptDescr

Exported constructors:

- Option :: String → [String] → (ArgDescr a) → String → OptDescr a

data ArgDescr

Exported constructors:

- NoArg :: a → ArgDescr a
- ReqArg :: (String → a) → String → ArgDescr a
- OptArg :: (Maybe String → a) → String → ArgDescr a

Exported functions:

usageInfo :: String → [OptDescr a] → String

getOpt :: ArgOrder a → [OptDescr a] → [String] → ([a], [String], [String])

getOpt' :: ArgOrder a → [OptDescr a] → [String] → ([a], [String], [String], [String])

A.2.19 Library Global

Library for handling global entities. A global entity has a name declared in the program. Its value can be accessed and modified by IO actions. Furthermore, global entities can be declared as persistent so that their values are stored across different program executions.

Currently, it is still experimental so that its interface might be slightly changed in the future.

A global entity `g` with an initial value `v` of type `t` must be declared by:

```
g :: Global t
g = global v spec
```

Here, the type `t` must not contain type variables and `spec` specifies the storage mechanism for the global entity (see type `GlobalSpec`).

Exported types:

```
data Global
```

The abstract type of a global entity.

Exported constructors:

```
data GlobalSpec
```

The storage mechanism for the global entity.

Exported constructors:

- `Temporary :: GlobalSpec`

`Temporary`

– the global value exists only during a single execution of a program

- `Persistent :: String → GlobalSpec`

`Persistent f`

– the global value is stored persistently in file `f` (which is created and initialized if it does not exist)

Exported functions:

```
global :: a → GlobalSpec → Global a
```

`global` is only used for the declaration of a global value and should not be used elsewhere. In the future, it might become a keyword.

```
readGlobal :: Global a → IO a
```

Reads the current value of a global.

```
writeGlobal :: Global a → a → IO ()
```

Updates the value of a global. The value is evaluated to a ground constructor term before it is updated.

A.2.20 Library GUI

This library contains definitions and functions to implement graphical user interfaces for Curry programs. It is based on Tcl/Tk and its basic ideas are described in detail [in this paper](#)

Exported types:

data GuiPort

The port to a GUI is just the stream connection to a GUI where Tcl/Tk communication is done.

Exported constructors:

data Widget

The type of possible widgets in a GUI.

Exported constructors:

- PlainButton :: [ConfItem] → Widget

PlainButton

– a button in a GUI whose event handler is activated if the user presses the button

- Canvas :: [ConfItem] → Widget

Canvas

– a canvas to draw pictures containing CanvasItems

- CheckButton :: [ConfItem] → Widget

CheckButton

– a check button: it has value "0" if it is unchecked and value "1" if it is checked

- Entry :: [ConfItem] → Widget

Entry

– an entry widget for entering single lines

- Label :: [ConfItem] → Widget

Label

– a label for showing a text

- ListBox :: [ConfItem] → Widget

ListBox

– a widget containing a list of items for selection

- `Message :: [ConfItem] → Widget`
`Message`
 - a message for showing simple string values
- `MenuButton :: [ConfItem] → Widget`
`MenuButton`
 - a button with a pull-down menu
- `Scale :: Int → Int → [ConfItem] → Widget`
`Scale`
 - a scale widget to input values by a slider
- `ScrollH :: WidgetRef → [ConfItem] → Widget`
`ScrollH`
 - a horizontal scroll bar
- `ScrollV :: WidgetRef → [ConfItem] → Widget`
`ScrollV`
 - a vertical scroll bar
- `TextEdit :: [ConfItem] → Widget`
`TextEdit`
 - a text editor widget to show and manipulate larger text paragraphs
- `Row :: [ConfCollection] → [Widget] → Widget`
`Row`
 - a horizontal alignment of widgets
- `Col :: [ConfCollection] → [Widget] → Widget`
`Col`
 - a vertical alignment of widgets
- `Matrix :: [ConfCollection] → [[Widget]] → Widget`
`Matrix`
 - a 2-dimensional (matrix) alignment of widgets

`data ConfItem`

The data type for possible configurations of a widget.

Exported constructors:

- `Active :: Bool → ConfItem`

`Active`

– define the active state for buttons, entries, etc.

- `Anchor :: String → ConfItem`

`Anchor`

– alignment of information inside a widget where the argument must be: n, ne, e, se, s, sw, w, nw, or center

- `Background :: String → ConfItem`

`Background`

– the background color

- `Foreground :: String → ConfItem`

`Foreground`

– the foreground color

- `Handler :: Event → (GuiPort → IO [ReconfigureItem]) → ConfItem`

`Handler`

– an event handler associated to a widget. The event handler returns a list of widget ref/configuration pairs that are applied after the handler in order to configure GUI widgets

- `Height :: Int → ConfItem`

`Height`

– the height of a widget (chars for text, pixels for graphics)

- `CheckInit :: String → ConfItem`

`CheckInit`

– initial value for checkbuttons

- `CanvasItems :: [CanvasItem] → ConfItem`

`CanvasItems`

– list of items contained in a canvas

- `List :: [String] → ConfItem`

`List`

– list of values shown in a listbox

- `Menu :: [MenuItem] → ConfItem`
`Menu`
 - the items of a menu button
- `WRef :: WidgetRef → ConfItem`
`WRef`
 - a reference to this widget
- `Text :: String → ConfItem`
`Text`
 - an initial text contents
- `Width :: Int → ConfItem`
`Width`
 - the width of a widget (chars for text, pixels for graphics)
- `Fill :: ConfItem`
`Fill`
 - fill widget in both directions
- `FillX :: ConfItem`
`FillX`
 - fill widget in horizontal direction
- `FillY :: ConfItem`
`FillY`
 - fill widget in vertical direction
- `TclOption :: String → ConfItem`
`TclOption`
 - further options in Tcl syntax (unsafe!)

`data ReconfigureItem`

Data type for describing configurations that are applied to a widget or GUI by some event handler.

Exported constructors:

- `WidgetConf :: WidgetRef → ConfItem → ReconfigureItem`
`WidgetConf wref conf`

– reconfigure the widget referred by wref with configuration item conf

- `StreamHandler :: Handle → (Handle → GuiPort → IO [ReconfigureItem]) → ReconfigureItem`

`StreamHandler hdl handler`

– add a new handler to the GUI that processes inputs on an input stream referred by hdl

- `RemoveStreamHandler :: Handle → ReconfigureItem`

`RemoveStreamHandler hdl`

– remove a handler for an input stream referred by hdl from the GUI (usually used to remove handlers for closed streams)

`data Event`

The data type of possible events on which handlers can react. This list is still incomplete and might be extended or restructured in future releases of this library.

Exported constructors:

- `DefaultEvent :: Event`

`DefaultEvent`

– the default event of the widget

- `MouseButton1 :: Event`

`MouseButton1`

– left mouse button pressed

- `MouseButton2 :: Event`

`MouseButton2`

– middle mouse button pressed

- `MouseButton3 :: Event`

`MouseButton3`

– right mouse button pressed

- `KeyPress :: Event`

`KeyPress`

– any key is pressed

- `Return :: Event`

`Return`

- return key is pressed

`data ConfCollection`

The data type for possible configurations of widget collections (e.g., columns, rows).

Exported constructors:

- `CenterAlign :: ConfCollection`

`CenterAlign`

- centered alignment

- `LeftAlign :: ConfCollection`

`LeftAlign`

- left alignment

- `RightAlign :: ConfCollection`

`RightAlign`

- right alignment

- `TopAlign :: ConfCollection`

`TopAlign`

- top alignment

- `BottomAlign :: ConfCollection`

`BottomAlign`

- bottom alignment

`data MenuItem`

The data type for specifying items in a menu.

Exported constructors:

- `MButton :: (GuiPort → IO [ReconfigureItem]) → String → MenuItem`

`MButton`

- a button with an associated command and a label string

- `MSeparator :: MenuItem`

`MSeparator`

- a separator between menu entries

- `MMenuButton :: String → [MenuItem] → MenuItem`
`MMenuButton`

– a submenu with a label string

`data CanvasItem`

The data type of items in a canvas. The last argument are further options in Tcl/Tk (for testing).

Exported constructors:

- `CLine :: [(Int,Int)] → String → CanvasItem`
- `CPolygon :: [(Int,Int)] → String → CanvasItem`
- `CRectangle :: (Int,Int) → (Int,Int) → String → CanvasItem`
- `C Oval :: (Int,Int) → (Int,Int) → String → CanvasItem`
- `CText :: (Int,Int) → String → String → CanvasItem`

`data WidgetRef`

The (hidden) data type of references to a widget in a GUI window. Note that the constructor `WRefLabel` will not be exported so that values can only be created inside this module.

Exported constructors:

`data Style`

The data type of possible text styles.

Exported constructors:

- `Bold :: Style`
`Bold`
– text in bold font
- `Italic :: Style`
`Italic`
– text in italic font
- `Underline :: Style`
`Underline`
– underline text

- `Fg :: Color → Style`

`Fg`

– foreground color, i.e., color of the text font

- `Bg :: Color → Style`

`Bg`

– background color of the text

`data Color`

The data type of possible colors.

Exported constructors:

- `Black :: Color`
- `Blue :: Color`
- `Brown :: Color`
- `Cyan :: Color`
- `Gold :: Color`
- `Gray :: Color`
- `Green :: Color`
- `Magenta :: Color`
- `Navy :: Color`
- `Orange :: Color`
- `Pink :: Color`
- `Purple :: Color`
- `Red :: Color`
- `Tomato :: Color`
- `Turquoise :: Color`
- `Violet :: Color`
- `White :: Color`
- `Yellow :: Color`

Exported functions:

`row :: [Widget] → Widget`

Horizontal alignment of widgets.

`col :: [Widget] → Widget`

Vertical alignment of widgets.

`matrix :: [[Widget]] → Widget`

Matrix alignment of widgets.

`debugTcl :: Widget → IO ()`

Prints the generated Tcl commands of a main widget (useful for debugging).

`runPassiveGUI :: String → Widget → IO GuiPort`

IO action to show a Widget in a new GUI window in passive mode, i.e., ignore all GUI events.

`runGUI :: String → Widget → IO ()`

IO action to run a Widget in a new window.

`runGUIwithParams :: String → String → Widget → IO ()`

IO action to run a Widget in a new window.

`runInitGUI :: String → Widget → (GuiPort → IO [ReconfigureItem]) → IO ()`

IO action to run a Widget in a new window. The GUI events are processed after executing an initial action on the GUI.

`runInitGUIwithParams :: String → String → Widget → (GuiPort → IO [ReconfigureItem]) → IO ()`

IO action to run a Widget in a new window. The GUI events are processed after executing an initial action on the GUI.

`runControlledGUI :: String → (Widget, String → GuiPort → IO ()) → Handle → IO ()`

Runs a Widget in a new GUI window and process GUI events. In addition, an event handler is provided that process messages received from an external stream identified by a handle (third argument). This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

`runConfigControlledGUI :: String → (Widget, String → GuiPort → IO [ReconfigureItem]) → Handle → IO ()`

Runs a Widget in a new GUI window and process GUI events. In addition, an event handler is provided that process messages received from an external stream identified by a handle (third argument). This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

```
runInitControlledGUI :: String → (Widget, String → GuiPort → IO ()) → (GuiPort  
→ IO [ReconfigureItem]) → Handle → IO ()
```

Runs a Widget in a new GUI window and process GUI events after executing an initial action on the GUI window. In addition, an event handler is provided that process messages received from an external message stream. This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

```
runHandlesControlledGUI :: String → (Widget, [Handle → GuiPort → IO  
[ReconfigureItem]]) → [Handle] → IO ()
```

Runs a Widget in a new GUI window and process GUI events. In addition, a list of event handlers is provided that process inputs received from a corresponding list of handles to input streams. Thus, if the i-th handle has some data available, the i-th event handler is executed with the i-th handle as a parameter. This operation is useful to run a GUI that should react on inputs provided by other processes, e.g., via sockets.

```
runInitHandlesControlledGUI :: String → (Widget, [Handle → GuiPort → IO  
[ReconfigureItem]]) → (GuiPort → IO [ReconfigureItem]) → [Handle] → IO ()
```

Runs a Widget in a new GUI window and process GUI events after executing an initial action on the GUI window. In addition, a list of event handlers is provided that process inputs received from a corresponding list of handles to input streams. Thus, if the i-th handle has some data available, the i-th event handler is executed with the i-th handle as a parameter. This operation is useful to run a GUI that should react on inputs provided by other processes, e.g., via sockets.

```
setConfig :: WidgetRef → ConfItem → GuiPort → IO ()
```

Changes the current configuration of a widget (deprecated operation, only included for backward compatibility). Warning: does not work for Command options!

```
exitGUI :: GuiPort → IO ()
```

An event handler for terminating the GUI.

```
getValue :: WidgetRef → GuiPort → IO String
```

Gets the (String) value of a variable in a GUI.

```
setValue :: WidgetRef → String → GuiPort → IO ()
```

Sets the (String) value of a variable in a GUI.

```
updateValue :: (String → String) → WidgetRef → GuiPort → IO ()
```

Updates the (String) value of a variable w.r.t. to an update function.

`appendValue :: WidgetRef → String → GuiPort → IO ()`

Appends a String value to the contents of a TextEdit widget and adjust the view to the end of the TextEdit widget.

`appendStyledValue :: WidgetRef → String → [Style] → GuiPort → IO ()`

Appends a String value with style tags to the contents of a TextEdit widget and adjust the view to the end of the TextEdit widget. Different styles can be combined, e.g., to get bold blue text on a red background. If **Bold**, **Italic** and **Underline** are combined, currently all but one of these are ignored. This is an experimental function and might be changed in the future.

`addRegionStyle :: WidgetRef → (Int,Int) → (Int,Int) → Style → GuiPort → IO ()`

Adds a style value in a region of a TextEdit widget. The region is specified a start and end position similarly to `getCursorPosition`. Different styles can be combined, e.g., to get bold blue text on a red background. If **Bold**, **Italic** and **Underline** are combined, currently all but one of these are ignored. This is an experimental function and might be changed in the future.

`removeRegionStyle :: WidgetRef → (Int,Int) → (Int,Int) → Style → GuiPort → IO ()`

Removes a style value in a region of a TextEdit widget. The region is specified a start and end position similarly to `getCursorPosition`. This is an experimental function and might be changed in the future.

`getCursorPosition :: WidgetRef → GuiPort → IO (Int,Int)`

Get the position (line,column) of the insertion cursor in a TextEdit widget. Lines are numbered from 1 and columns are numbered from 0.

`seeText :: WidgetRef → (Int,Int) → GuiPort → IO ()`

Adjust the view of a TextEdit widget so that the specified line/column character is visible. Lines are numbered from 1 and columns are numbered from 0.

`focusInput :: WidgetRef → GuiPort → IO ()`

Sets the input focus of this GUI to the widget referred by the first argument. This is useful for automatically selecting input entries in an application.

`addCanvas :: WidgetRef → [CanvasItem] → GuiPort → IO ()`

Adds a list of canvas items to a canvas referred by the first argument.

`popupMessage :: String → IO ()`

A simple popup message.

`Cmd :: (GuiPort → IO ()) → ConfItem`

A simple event handler that can be associated to a widget. The event handler takes a GUI port as parameter in order to read or write values from/into the GUI.

`Command :: (GuiPort → IO [ReconfigureItem]) → ConfItem`

An event handler that can be associated to a widget. The event handler takes a GUI port as parameter (in order to read or write values from/into the GUI) and returns a list of widget reference/configuration pairs which is applied after the handler in order to configure some GUI widgets.

`Button :: (GuiPort → IO ()) → [ConfItem] → Widget`

A button with an associated event handler which is activated if the button is pressed.

`ConfigButton :: (GuiPort → IO [ReconfigureItem]) → [ConfItem] → Widget`

A button with an associated event handler which is activated if the button is pressed. The event handler is a configuration handler (see `Command`) that allows the configuration of some widgets.

`TextEditScroll :: [ConfItem] → Widget`

A text edit widget with vertical and horizontal scrollbars. The argument contains the configuration options for the text edit widget.

`ListBoxScroll :: [ConfItem] → Widget`

A list box widget with vertical and horizontal scrollbars. The argument contains the configuration options for the list box widget.

`CanvasScroll :: [ConfItem] → Widget`

A canvas widget with vertical and horizontal scrollbars. The argument contains the configuration options for the text edit widget.

`EntryScroll :: [ConfItem] → Widget`

An entry widget with a horizontal scrollbar. The argument contains the configuration options for the entry widget.

`getOpenFile :: IO String`

Pops up a GUI for selecting an existing file. The file with its full path name will be returned (or "" if the user cancels the selection).

`getOpenFileWithTypes :: [(String,String)] → IO String`

Pops up a GUI for selecting an existing file. The parameter is a list of pairs of file types that could be selected. A file type pair consists of a name and an extension for that file type. The file with its full path name will be returned (or "" if the user cancels the selection).

`getSaveFile :: IO String`

Pops up a GUI for choosing a file to save some data. If the user chooses an existing file, she/he will be asked to confirm to overwrite it. The file with its full path name will be returned (or "" if the user cancels the selection).

`getSaveFileWithTypes :: [(String,String)] → IO String`

Pops up a GUI for choosing a file to save some data. The parameter is a list of pairs of file types that could be selected. A file type pair consists of a name and an extension for that file type. If the user chooses an existing file, she/he will be asked to confirm to overwrite it. The file with its full path name will be returned (or "" if the user cancels the selection).

`chooseColor :: IO String`

Pops up a GUI dialog box to select a color. The name of the color will be returned (or "" if the user cancels the selection).

A.2.21 Library Integer

A collection of common operations on integer numbers. Most operations make no assumption on the precision of integers. Operation `bitNot` is necessarily an exception.

Exported functions:

`(^) :: Int → Int → Int`

The value of `a ^ b` is `a` raised to the power of `b`. Fails if `b < 0`. Executes in $O(\log b)$ steps.

`pow :: Int → Int → Int`

The value of `pow a b` is `a` raised to the power of `b`. Fails if `b < 0`. Executes in $O(\log b)$ steps.

`ilog :: Int → Int`

The value of `ilog n` is the floor of the logarithm in the base 10 of `n`. Fails if `n <= 0`. For positive integers, the returned value is 1 less the number of digits in the decimal representation of `n`.

`isqrt :: Int → Int`

The value of `isqrt n` is the floor of the square root of `n`. Fails if `n < 0`. Executes in $O(\log n)$ steps, but there must be a better way.

`factorial :: Int → Int`

The value of `factorial n` is the factorial of `n`. Fails if `n < 0`.

`binomial :: Int → Int → Int`

The value of `binomial n m` is $n(n-1)\dots(n-m+1)/m(m-1)\dots 1$. Fails if ‘`m <= 0`’ or ‘`n < m`’.

`abs :: Int → Int`

The value of `abs n` is the absolute value of `n`.

`max3 :: a → a → a → a`

Returns the maximum of the three arguments.

`min3 :: a → a → a → a`

Returns the minimum of the three arguments.

`maxlist :: [a] → a`

Returns the maximum of a list of integer values. Fails if the list is empty.

`minlist :: [a] → a`

Returns the minimum of a list of integer values. Fails if the list is empty.

`bitTrunc :: Int → Int → Int`

The value of `bitTrunc n m` is the value of the `n` least significant bits of `m`.

`bitAnd :: Int → Int → Int`

Returns the bitwise AND of the two arguments.

`bitOr :: Int → Int → Int`

Returns the bitwise inclusive OR of the two arguments.

`bitNot :: Int → Int`

Returns the bitwise NOT of the argument. Since integers have unlimited precision, only the 32 least significant bits are computed.

`bitXor :: Int → Int → Int`

Returns the bitwise exclusive OR of the two arguments.

`even :: Int → Bool`

Returns whether an integer is even

`odd :: Int → Bool`

Returns whether an integer is odd

A.2.22 Library IO

Library for IO operations like reading and writing files that are not already contained in the prelude.

Exported types:

`data Handle`

The abstract type of a handle for a stream.

Exported constructors:

`data IOMode`

The modes for opening a file.

Exported constructors:

- `ReadMode :: IOMode`
- `WriteMode :: IOMode`
- `AppendMode :: IOMode`

`data SeekMode`

The modes for positioning with `hSeek` in a file.

Exported constructors:

- `AbsoluteSeek :: SeekMode`
- `RelativeSeek :: SeekMode`
- `SeekFromEnd :: SeekMode`

Exported functions:

`stdin :: Handle`

Standard input stream.

`stdout :: Handle`

Standard output stream.

`stderr :: Handle`

Standard error stream.

`openFile :: String → IOMode → IO Handle`

Opens a file in specified mode and returns a handle to it.

`hClose :: Handle → IO ()`

Closes a file handle and flushes the buffer in case of output file.

`hFlush :: Handle → IO ()`

Flushes the buffer associated to handle in case of output file.

`hIsEOF :: Handle → IO Bool`

Is handle at end of file?

`isEOF :: IO Bool`

Is standard input at end of file?

`hSeek :: Handle → SeekMode → Int → IO ()`

Set the position of a handle to a seekable stream (e.g., a file). If the second argument is `AbsoluteSeek`, `SeekFromEnd`, or `RelativeSeek`, the position is set relative to the beginning of the file, to the end of the file, or to the current position, respectively.

`hWaitForInput :: Handle → Int → IO Bool`

Waits until input is available on the given handle. If no input is available within `t` milliseconds, it returns `False`, otherwise it returns `True`.

`hWaitForInputs :: [Handle] → Int → IO Int`

Waits until input is available on some of the given handles. If no input is available within `t` milliseconds, it returns `-1`, otherwise it returns the index of the corresponding handle with the available data.

`hWaitForInputOrMsg :: Handle → [a] → IO (Either Handle [a])`

Waits until input is available on a given handles or a message in the message stream. Usually, the message stream comes from an external port. Thus, this operation implements a committed choice over receiving input from an IO handle or an external port.

Note that the implementation of this operation works only with Sicstus-Prolog 3.8.5 or higher (due to a bug in previous versions of Sicstus-Prolog).

`hWaitForInputsOrMsg :: [Handle] → [a] → IO (Either Int [a])`

Waits until input is available on some of the given handles or a message in the message stream. Usually, the message stream comes from an external port. Thus, this operation implements a committed choice over receiving input from IO handles or an external port.

Note that the implementation of this operation works only with Sicstus-Prolog 3.8.5 or higher (due to a bug in previous versions of Sicstus-Prolog).

`hReady :: Handle → IO Bool`

Checks whether an input is available on a given handle.

`hGetChar :: Handle → IO Char`

Reads a character from an input handle and returns it. Throws an error if the end of file has been reached.

`hGetLine :: Handle → IO String`

Reads a line from an input handle and returns it. Throws an error if the end of file has been reached while reading the *first* character. If the end of file is reached later in the line, it is treated as a line terminator and the (partial) line is returned.

`hGetContents :: Handle → IO String`

Reads the complete contents from an input handle and closes the input handle before returning the contents.

`getContents :: IO String`

Reads the complete contents from the standard input stream until EOF.

`hPutChar :: Handle → Char → IO ()`

Puts a character to an output handle.

`hPutStr :: Handle → String → IO ()`

Puts a string to an output handle.

`hPutStrLn :: Handle → String → IO ()`

Puts a string with a newline to an output handle.

`hPrint :: Handle → a → IO ()`

Converts a term into a string and puts it to an output handle.

`hIsReadable :: Handle → IO Bool`

Is the handle readable?

`hIsWritable :: Handle → IO Bool`

Is the handle writable?

`hIsTerminalDevice :: Handle → IO Bool`

Is the handle connected to a terminal?

A.2.23 Library IOExts

Library with some useful extensions to the IO monad.

Exported types:

`data IORef`

Mutable variables containing values of some type. The values are not evaluated when they are assigned to an `IORef`.

Exported constructors:

Exported functions:

`execCmd :: String → IO (Handle,Handle,Handle)`

Executes a command with a new default shell process. The standard I/O streams of the new process (`stdin,stdout,stderr`) are returned as handles so that they can be explicitly manipulated. They should be closed with `IO.hClose` since they are not closed automatically when the process terminates.

`evalCmd :: String → [String] → String → IO (Int,String,String)`

Executes a command with the given arguments as a new default shell process and provides the input via the process' `stdin` input stream. The exit code of the process and the contents written to the standard I/O streams `stdout` and `stderr` are returned.

`connectToCommand :: String → IO Handle`

Executes a command with a new default shell process. The input and output streams of the new process is returned as one handle which is both readable and writable. Thus, writing to the handle produces input to the process and output from the process can be retrieved by reading from this handle. The handle should be closed with `IO.hClose` since they are not closed automatically when the process terminates.

`readCompleteFile :: String → IO String`

An action that reads the complete contents of a file and returns it. This action can be used instead of the (lazy) `readFile` action if the contents of the file might be changed.

`updateFile :: (String → String) → String → IO ()`

An action that updates the contents of a file.

`exclusiveIO :: String → IO a → IO a`

Forces the exclusive execution of an action via a lock file. For instance, (`exclusiveIO "myaction.lock" act`) ensures that the action "act" is not executed by two processes on the same system at the same time.

`setAssoc :: String → String → IO ()`

Defines a global association between two strings. Both arguments must be evaluable to ground terms before applying this operation.

`getAssoc :: String → IO (Maybe String)`

Gets the value associated to a string. Nothing is returned if there does not exist an associated value.

`newIORef :: a → IO (IORef a)`

Creates a new IORef with an initial values.

`readIORef :: IORef a → IO a`

Reads the current value of an IORef.

`writeIORef :: IORef a → a → IO ()`

Updates the value of an IORef.

`modifyIORef :: IORef a → (a → a) → IO ()`

Modify the value of an IORef.

A.2.24 Library JavaScript

A library to represent JavaScript programs.

Exported types:

`data JSExp`

Type of JavaScript expressions.

Exported constructors:

- `JSSString :: String → JSExp`

`JSSString`

– string constant

- `JSInt :: Int → JSExp`

`JSInt`

– integer constant

- `JSBool :: Bool → JSExp`

`JSBool`

– Boolean constant

- `JSIVar :: Int → JSExp`

`JSIVar`

– indexed variable

- `JSIArrayIdx :: Int → Int → JSExp`
`JSIArrayIdx`
 – array access to index array variable
- `JSOp :: String → JSExp → JSExp → JSExp`
`JSOp`
 – infix operator expression
- `JSFCall :: String → [JSExp] → JSExp`
`JSFCall`
 – function call
- `JSApply :: JSExp → JSExp → JSExp`
`JSApply`
 – function call where the function is an expression
- `JSLambda :: [Int] → [JSStat] → JSExp`
`JSLambda`
 – (anonymous) function with indexed variables as arguments

data JSStat

Type of JavaScript statements.

Exported constructors:

- `JSAssign :: JSExp → JSExp → JSStat`
`JSAssign`
 – assignment
- `JSIf :: JSExp → [JSStat] → [JSStat] → JSStat`
`JSIf`
 – conditional
- `JSSwitch :: JSExp → [JSBranch] → JSStat`
`JSSwitch`
 – switch statement
- `JSPCall :: String → [JSExp] → JSStat`
`JSPCall`

- procedure call
- JSReturn :: JSExp → JSStat
JSReturn
 - return statement
- JSVarDecl :: Int → JSStat
JSVarDecl
 - local variable declaration

data JSBranch

Exported constructors:

- JSCase :: String → [JSStat] → JSBranch
JSCase
 - case branch
- JSDefault :: [JSStat] → JSBranch
JSDefault
 - default branch

data JSFDecl

Exported constructors:

- JSFDecl :: String → [Int] → [JSStat] → JSFDecl

Exported functions:

showJSExp :: JSExp → String

Shows a JavaScript expression as a string in JavaScript syntax.

showJSStat :: Int → JSStat → String

Shows a JavaScript statement as a string in JavaScript syntax with indenting.

showJSFDecl :: JSFDecl → String

Shows a JavaScript function declaration as a string in JavaScript syntax.

jsConsTerm :: String → [JSExp] → JSExp

Representation of constructor terms in JavaScript.

A.2.25 Library `KeyDatabaseSQLite`

This module provides a general interface for databases (persistent predicates) where each entry consists of a key and an info part. The key is an integer and the info is arbitrary. All functions are parameterized with a dynamic predicate that takes an integer key as a first parameter.

This module reimplements the interface of the module `KeyDatabase` based on the `SQLite` database engine. In order to use it you need to have `sqlite3` in your `PATH` environment variable or adjust the value of the constant `path<code>to</code>sqlite3`.

Programs that use the `KeyDatabase` module can be adjusted to use this module instead by replacing the imports of `Dynamic`, `Database`, and `KeyDatabase` with this module and changing the declarations of database predicates to use the function `persistentSQLite` instead of `dynamic` or `persistent`. This module redefines the types `Dynamic`, `Query`, and `Transaction` and although both implementations can be used in the same program (by importing modules qualified) they cannot be mixed.

Compared with the interface of `KeyDatabase`, this module lacks definitions for `index`, `sortByIndex`, `groupByIndex`, and `runTNA` and adds the functions `deleteDBEntries` and `closeDBHandles`.

Exported types:

```
type Key = Int
```

```
type KeyPred a = Int → a → Dynamic
```

```
data Query
```

Queries can read but not write to the database.

Exported constructors:

```
data Transaction
```

Transactions can modify the database and are executed atomically.

Exported constructors:

```
data Dynamic
```

Result type of database predicates.

Exported constructors:

```
data ColVal
```

Abstract type for value restrictions

Exported constructors:

`data TError`

The type of errors that might occur during a transaction.

Exported constructors:

- `TError :: TErrorKind → String → TError`

`data TErrorKind`

The various kinds of transaction errors.

Exported constructors:

- `KeyNotExistsError :: TErrorKind`
- `NoRelationshipError :: TErrorKind`
- `DuplicateKeyError :: TErrorKind`
- `KeyRequiredError :: TErrorKind`
- `UniqueError :: TErrorKind`
- `MinError :: TErrorKind`
- `MaxError :: TErrorKind`
- `UserDefinedError :: TErrorKind`
- `ExecutionError :: TErrorKind`

Exported functions:

`runQ :: Query a → IO a`

Runs a database query in the IO monad.

`transformQ :: (a → b) → Query a → Query b`

Applies a function to the result of a database query.

`runT :: Transaction a → IO (Either a TError)`

Runs a transaction atomically in the IO monad.

Transactions are *immediate*, which means that locks are acquired on all databases as soon as the transaction is started. After one transaction is started, no other database connection will be able to write to the database or start a transaction. Other connections *can* read the database during a transaction of another process.

The choice to use immediate rather than deferred transactions is conservative. It might also be possible to allow multiple simultaneous transactions that lock tables on the first database access (which is the default in SQLite). However this leads to unpredictable

order in which locks are taken when multiple databases are involved. The current implementation fixes the locking order by sorting databases by their name and locking them in order immediately when a transaction begins.

More information on ⁶ `_transaction.html`>transactions in SQLite is available online.

`runJustT :: Transaction a → IO a`

Executes a possibly composed transaction on the current state of dynamic predicates as a single transaction. Similar to `runT` but a run-time error is raised if the execution of the transaction fails.

`getDB :: Query a → Transaction a`

Lifts a database query to the transaction type such that it can be composed with other transactions. Run-time errors that occur during the execution of the given query are transformed into transaction errors.

`returnT :: a → Transaction a`

Returns the given value in a transaction that does not access the database.

`doneT :: Transaction ()`

Returns the unit value in a transaction that does not access the database. Useful to ignore results when composing transactions.

`errorT :: TError → Transaction a`

Aborts a transaction with an error.

`failT :: String → Transaction a`

Aborts a transaction with a user-defined error message.

`(|>>=) :: Transaction a → (a → Transaction b) → Transaction b`

Combines two transactions into a single transaction that executes both in sequence. The first transaction is executed, its result passed to the function which computes the second transaction, which is then executed to compute the final result.

If the first transaction is aborted with an error, the second transaction is not executed.

`(|>>) :: Transaction a → Transaction b → Transaction b`

Combines two transactions to execute them in sequence. The result of the first transaction is ignored.

`sequenceT :: [Transaction a] → Transaction [a]`

Executes a list of transactions sequentially and computes a list of all results.

⁶<http://sqlite.org/lang>

`sequenceT_ :: [Transaction a] → Transaction ()`

Executes a list of transactions sequentially, ignoring their results.

`mapT :: (a → Transaction b) → [a] → Transaction [b]`

Applies a function that yields transactions to all elements of a list, executes the transaction sequentially, and collects their results.

`mapT_ :: (a → Transaction b) → [a] → Transaction ()`

Applies a function that yields transactions to all elements of a list, executes the transactions sequentially, and ignores their results.

`persistentSQLite :: String → String → [String] → Int → a → Dynamic`

This function is used instead of `dynamic` or `persistent` to declare predicates whose facts are stored in an SQLite database.

If the provided database or the table do not exist they are created automatically when the declared predicate is accessed for the first time.

Multiple column names can be provided if the second argument of the predicate is a tuple with a matching arity. Other record types are not supported. If no column names are provided a table with a single column called `info` is created. Columns of name `rowid` are not supported and lead to a run-time error.

`existsDBKey :: (Int → a → Dynamic) → Int → Query Bool`

Checks whether the predicate has an entry with the given key.

`allDBKeys :: (Int → a → Dynamic) → Query [Int]`

Returns a list of all stored keys. Do not use this function unless the database is small.

`allDBInfos :: (Int → a → Dynamic) → Query [a]`

Returns a list of all info parts of stored entries. Do not use this function unless the database is small.

`allDBKeyInfos :: (Int → a → Dynamic) → Query [(Int,a)]`

Returns a list of all stored entries. Do not use this function unless the database is small.

`(@=) :: Int → a → ColVal`

Constructs a value restriction for the column given as first argument

`someDBKeys :: (Int → a → Dynamic) → [ColVal] → Query [Int]`

Returns a list of those stored keys where the corresponding info part matches the given value restriction. Safe to use even on large databases if the number of results is small.

`someDBInfos :: (Int → a → Dynamic) → [ColVal] → Query [a]`

Returns a list of those info parts of stored entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

```
someDBKeyInfos :: (Int → a → Dynamic) → [ColVal] → Query [(Int,a)]
```

Returns a list of those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

```
someDBKeyProjections :: (Int → a → Dynamic) → [Int] → [ColVal] → Query [(Int,b)]
```

Returns a list of column projections on those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

```
getDBInfo :: (Int → a → Dynamic) → Int → Query (Maybe a)
```

Queries the information stored under the given key. Yields `Nothing` if the given key is not present.

```
getDBInfos :: (Int → a → Dynamic) → [Int] → Query (Maybe [a])
```

Queries the information stored under the given keys. Yields `Nothing` if a given key is not present.

```
deleteDBEntry :: (Int → a → Dynamic) → Int → Transaction ()
```

Deletes the information stored under the given key. If the given key does not exist this transaction is silently ignored and no error is raised.

```
deleteDBEntries :: (Int → a → Dynamic) → [Int] → Transaction ()
```

Deletes the information stored under the given keys. No error is raised if (some of) the keys do not exist.

```
updateDBEntry :: (Int → a → Dynamic) → Int → a → Transaction ()
```

Updates the information stored under the given key. The transaction is aborted with a `KeyNotExistsError` if the given key is not present in the database.

```
newDBEntry :: (Int → a → Dynamic) → a → Transaction Int
```

Stores new information in the database and yields the newly generated key.

```
newDBKeyEntry :: (Int → a → Dynamic) → Int → a → Transaction ()
```

Stores a new entry in the database under a given key. The transaction fails if the key already exists.

```
cleanDB :: (Int → a → Dynamic) → Transaction ()
```

Deletes all entries from the database associated with a predicate.

```
closeDBHandles :: IO ()
```

Closes all database connections. Should be called when no more database access will be necessary.

```
showTError :: TError → String
```

Transforms a transaction error into a string.

A.2.26 Library List

Library with some useful operations on lists.

Exported functions:

`elemIndex :: a → [a] → Maybe Int`

Returns the index `i` of the first occurrence of an element in a list as `(Just i)`, otherwise `Nothing` is returned.

`elemIndices :: a → [a] → [Int]`

Returns the list of indices of occurrences of an element in a list.

`find :: (a → Bool) → [a] → Maybe a`

Returns the first element `e` of a list satisfying a predicate as `(Just e)`, otherwise `Nothing` is returned.

`findIndex :: (a → Bool) → [a] → Maybe Int`

Returns the index `i` of the first occurrences of a list element satisfying a predicate as `(Just i)`, otherwise `Nothing` is returned.

`findIndices :: (a → Bool) → [a] → [Int]`

Returns the list of indices of list elements satisfying a predicate.

`nub :: [a] → [a]`

Removes all duplicates in the argument list.

`nubBy :: (a → a → Bool) → [a] → [a]`

Removes all duplicates in the argument list according to an equivalence relation.

`delete :: a → [a] → [a]`

Deletes the first occurrence of an element in a list.

`deleteBy :: (a → a → Bool) → a → [a] → [a]`

Deletes the first occurrence of an element in a list according to an equivalence relation.

`(\\) :: [a] → [a] → [a]`

Computes the difference of two lists.

`union :: [a] → [a] → [a]`

Computes the union of two lists.

`unionBy :: (a → a → Bool) → [a] → [a] → [a]`

Computes the union of two lists according to the given equivalence relation

```
intersect :: [a] → [a] → [a]
```

Computes the intersection of two lists.

```
intersectBy :: (a → a → Bool) → [a] → [a] → [a]
```

Computes the intersection of two lists according to the given equivalence relation

```
intersperse :: a → [a] → [a]
```

Puts a separator element between all elements in a list.

Example: `(intersperse 9 [1,2,3,4]) = [1,9,2,9,3,9,4]`

```
intercalate :: [a] → [[a]] → [a]
```

`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
transpose :: [[a]] → [[a]]
```

Transposes the rows and columns of the argument.

Example: `(transpose [[1,2,3],[4,5,6]]) = [[1,4],[2,5],[3,6]]`

```
permutations :: [a] → [[a]]
```

Returns the list of all permutations of the argument.

```
partition :: (a → Bool) → [a] → ([a],[a])
```

Partitions a list into a pair of lists where the first list contains those elements that satisfy the predicate argument and the second list contains the remaining arguments.

Example: `(partition (<4) [8,1,5,2,4,3]) = ([1,2,3],[8,5,4])`

```
group :: [a] → [[a]]
```

Splits the list argument into a list of lists of equal adjacent elements.

Example: `(group [1,2,2,3,3,3,4]) = [[1],[2,2],[3,3,3],[4]]`

```
groupBy :: (a → a → Bool) → [a] → [[a]]
```

Splits the list argument into a list of lists of related adjacent elements.

```
splitOn :: [a] → [a] → [[a]]
```

Breaks the second list argument into pieces separated by the first list argument, consuming the delimiter. An empty delimiter is invalid, and will cause an error to be raised.

```
split :: (a → Bool) → [a] → [[a]]
```

Splits a list into components delimited by separators, where the predicate returns True for a separator element. The resulting components do not contain the separators. Two adjacent separators result in an empty component in the output.

```
split (==a) "aabbaca" == ["", "", "bb", "c", ""] split (==a) "" == [""]
```

```
inits :: [a] → [[a]]
```

Returns all initial segments of a list, starting with the shortest. Example: `inits [1,2,3] == [[], [1], [1,2], [1,2,3]]`

```
tails :: [a] → [[a]]
```

Returns all final segments of a list, starting with the longest. Example: `tails [1,2,3] == [[1,2,3], [2,3], [3], []]`

```
replace :: a → Int → [a] → [a]
```

Replaces an element in a list.

```
isPrefixOf :: [a] → [a] → Bool
```

Checks whether a list is a prefix of another.

```
isSuffixOf :: [a] → [a] → Bool
```

Checks whether a list is a suffix of another.

```
isInfixOf :: [a] → [a] → Bool
```

Checks whether a list is contained in another.

```
sortBy :: (a → a → Bool) → [a] → [a]
```

Sorts a list w.r.t. an ordering relation by the insertion method.

```
insertBy :: (a → a → Bool) → a → [a] → [a]
```

Inserts an object into a list according to an ordering relation.

```
last :: [a] → a
```

Returns the last element of a non-empty list.

```
init :: [a] → [a]
```

Returns the input list with the last element removed.

```
sum :: [Int] → Int
```

Returns the sum of a list of integers.

```
product :: [Int] → Int
```

Returns the product of a list of integers.

`maximum :: [a] → a`

Returns the maximum of a non-empty list.

`maximumBy :: (a → a → Ordering) → [a] → a`

Returns the maximum of a non-empty list according to the given comparison function

`minimum :: [a] → a`

Returns the minimum of a non-empty list.

`minimumBy :: (a → a → Ordering) → [a] → a`

Returns the minimum of a non-empty list according to the given comparison function

`scanl :: (a → b → a) → a → [b] → [a]`

`scanl` is similar to `foldl`, but returns a list of successive reduced values from the left:

`scanl f z [x1, x2, ...] == [z, z f x1, (z f x1) f x2, ...]`

`scanl1 :: (a → a → a) → [a] → [a]`

`scanl1` is a variant of `scanl` that has no starting value argument: `scanl1 f [x1, x2, ...]`

`== [x1, x1 f x2, ...]`

`scanr :: (a → b → b) → b → [a] → [b]`

`scanr` is the right-to-left dual of `scanl`.

`scanr1 :: (a → a → a) → [a] → [a]`

`scanr1` is a variant of `scanr` that has no starting value argument.

`mapAccumL :: (a → b → (a,c)) → a → [b] → (a,[c])`

The `mapAccumL` function behaves like a combination of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

`mapAccumR :: (a → b → (a,c)) → a → [b] → (a,[c])`

The `mapAccumR` function behaves like a combination of `map` and `foldr`; it applies a function to each element of a list, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new list.

`cycle :: [a] → [a]`

Builds an infinite list from a finite one.

`unfoldr :: (a → Maybe (b,a)) → a → [b]`

Builds a list from a seed value.

A.2.27 Library Maybe

Library with some useful functions on the `Maybe` datatype.

Exported functions:

`isJust :: Maybe a → Bool`

Return `True` iff the argument is of the form `Just ..`

`isNothing :: Maybe a → Bool`

Return `True` iff the argument is of the form `Nothing`.

`fromJust :: Maybe a → a`

Extract the argument from the `Just` constructor and throw an error if the argument is `Nothing`.

`fromMaybe :: a → Maybe a → a`

Extract the argument from the `Just` constructor or return the provided default value if the argument is `Nothing`.

`listToMaybe :: [a] → Maybe a`

Return `Nothing` on an empty list or `Just x` where `x` is the first list element.

`maybeToList :: Maybe a → [a]`

Return an empty list for `Nothing` or a singleton list for `Just x`.

`catMaybes :: [Maybe a] → [a]`

Return the list of all `Just` values.

`mapMaybe :: (a → Maybe b) → [a] → [b]`

Apply a function which may throw out elements using the `Nothing` constructor to a list of elements.

`(>>-) :: Maybe a → (a → Maybe b) → Maybe b`

Monadic bind for `Maybe`. `Maybe` can be interpreted as a monad where `Nothing` is interpreted as the error case by this monadic binding.

`sequenceMaybe :: [Maybe a] → Maybe [a]`

Monadic sequence for `Maybe`.

`mapMMaybe :: (a → Maybe b) → [a] → Maybe [b]`

Monadic map for `Maybe`.

`mplus :: Maybe a → Maybe a → Maybe a`

Combine two `Maybe`s, returning the first `Just` value, if any.

A.2.28 Library NamedSocket

Library to support network programming with sockets that are addressed by symbolic names. In contrast to raw sockets (see library `Socket`), this library uses the Curry Port Name Server to provide sockets that are addressed by symbolic names rather than numbers.

In standard applications, the server side uses the operations `listenOn` and `socketAccept` to provide some service on a named socket, and the client side uses the operation `connectToSocket` to request a service.

Exported types:

`data Socket`

Abstract type for named sockets.

Exported constructors:

Exported functions:

`listenOn :: String → IO Socket`

Creates a server side socket with a symbolic name.

`socketAccept :: Socket → IO (String,Handle)`

Returns a connection of a client to a socket. The connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client. The handle is both readable and writable.

`waitForSocketAccept :: Socket → Int → IO (Maybe (String,Handle))`

Waits until a connection of a client to a socket is available. If no connection is available within the time limit, it returns `Nothing`, otherwise the connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client.

`sClose :: Socket → IO ()`

Closes a server socket.

`socketName :: Socket → String`

Returns a the symbolic name of a named socket.

`connectToSocketRepeat :: Int → IO a → Int → String → IO (Maybe Handle)`

Waits for connection to a Unix socket with a symbolic name. In contrast to `connectToSocket`, this action waits until the socket has been registered with its symbolic name.

`connectToSocketWait :: String → IO Handle`

Waits for connection to a Unix socket with a symbolic name and return the handle of the connection. This action waits (possibly forever) until the socket with the symbolic name is registered.

`connectToSocket :: String → IO Handle`

Creates a new connection to an existing(!) Unix socket with a symbolic name. If the symbolic name is not registered, an error is reported.

A.2.29 Library Parser

Library with functional logic parser combinators.

Adapted from: Rafael Caballero and Francisco J. Lopez-Fraguas: A Functional Logic Perspective of Parsing. In Proc. FLOPS'99, Springer LNCS 1722, pp. 85-99, 1999

Exported types:

`type Parser a = [a] → [a]`

`type ParserRep a b = a → [b] → [b]`

Exported functions:

`(<|>) :: ([a] → [a]) → ([a] → [a]) → [a] → [a]`

Combines two parsers without representation in an alternative manner.

`(<||>) :: (a → [b] → [b]) → (a → [b] → [b]) → a → [b] → [b]`

Combines two parsers with representation in an alternative manner.

`(<*>) :: ([a] → [a]) → ([a] → [a]) → [a] → [a]`

Combines two parsers (with or without representation) in a sequential manner.

`(>>>) :: ([a] → [a]) → b → b → [a] → [a]`

Attaches a representation to a parser without representation.

`empty :: [a] → [a]`

The empty parser which recognizes the empty word.

`terminal :: a → [a] → [a]`

A parser recognizing a particular terminal symbol.

`satisfy :: (a → Bool) → a → [a] → [a]`

A parser (with representation) recognizing a terminal satisfying a given predicate.

`star :: (a → [b] → [b]) → [a] → [b] → [b]`

A star combinator for parsers. The returned parser repeats zero or more times a parser `p` with representation and returns the representation of all parsers in a list.

`some :: (a → [b] → [b]) → [a] → [b] → [b]`

A some combinator for parsers. The returned parser repeats the argument parser (with representation) at least once.

A.2.30 Library Pretty

This library provides pretty printing combinators. The interface is that of [Daan Leijen's library linear-time, bounded implementation](#) by Olaf Chitil. Note that the implementation of `fill` and `fillBreak` is not linear-time bounded Support of ANSI escape codes for formatting and colorisation of documents in text terminals (see https://en.wikipedia.org/wiki/ANSI_escapecode)

Exported types:

`data Doc`

The abstract data type `Doc` represents pretty documents.

Exported constructors:

Exported functions:

`pPrint :: Doc → String`

Standard printing with a column length of 80.

`empty :: Doc`

The empty document

`isEmpty :: Doc → Bool`

Is the document empty?

`text :: String → Doc`

The document (`text s`) contains the literal string `s`. The string shouldn't contain any newline (`\n`) characters. If the string contains newline characters, the function `string` should be used.

`linesep :: String → Doc`

The document (`linesep s`) advances to the next line and indents to the current nesting level. Document (`linesep s`) behaves like (`text s`) if the line break is undone by `group`.

`hardline :: Doc`

The document `hardline` advances to the next line and indents to the current nesting level. `hardline` cannot be undone by `group`.

`line :: Doc`

The document `line` advances to the next line and indents to the current nesting level. Document `line` behaves like (`text " "`) if the line break is undone by `group`.

`linebreak :: Doc`

The document `linebreak` advances to the next line and indents to the current nesting level. Document `linebreak` behaves like (`text ""`) if the line break is undone by `group`.

`softline :: Doc`

The document `softline` behaves like `space` if the resulting output fits the page, otherwise it behaves like `line`. `softline = group line`

`softbreak :: Doc`

The document `softbreak` behaves like (`text ""`) if the resulting output fits the page, otherwise it behaves like `line`. `softbreak = group linebreak`

`group :: Doc → Doc`

The combinator `group` is used to specify alternative layouts. The document (`group x`) undoes all line breaks in document `x`. The resulting line is added to the current line if that fits the page. Otherwise, the document `x` is rendered without any changes.

`nest :: Int → Doc → Doc`

The document (`nest i d`) renders document `d` with the current indentation level increased by `i` (See also `hang`, `align` and `indent`).

```
nest 2 (text "hello" $$ text "world") $$ text "!"
```

outputs as:

```
hello
  world
!
```

`hang :: Int → Doc → Doc`

The combinator `hang` implements hanging indentation. The document `(hang i d)` renders document `d` with a nesting level set to the current column plus `i`. The following example uses hanging indentation for some text:

```
test = hang 4
      (fillSep
       (map text
        (words "the hang combinator indents these words !"))))
```

Which lays out on a page with a width of 20 characters as:

```
the hang combinator
  indents these
    words !
```

The `hang` combinator is implemented as:

```
hang i x = align (nest i x)
```

```
align :: Doc → Doc
```

The document `(align d)` renders document `d` with the nesting level set to the current column. It is used for example to implement `hang`.

As an example, we will put a document right above another one, regardless of the current nesting level:

```
x $$ y = align (x $$ y)
test   = text "hi" <+> (text "nice" $$ text "world")
```

which will be layed out as:

```
hi nice
  world
```

```
indent :: Int → Doc → Doc
```

The document `(indent i d)` indents document `d` with `i` spaces.

```
test = indent 4 (fillSep (map text
                          (words "the indent combinator indents these words !")))
```

Which lays out with a page width of 20 as:

the indent
combinator
indents these
words !

`combine :: Doc → Doc → Doc → Doc`

The document `(combine c d1 d2)` combines document `d1` and `d2` with document `c` in between using `(<>)` with identity `empty`. Thus, the following equations hold.

```
combine c d1    empty == d1
combine c empty d2  == d2
combine c d1    d2    == d1 <> c <> d2 if neither d1 nor d2 are empty
```

`(<>) :: Doc → Doc → Doc`

The document `(x <> y)` concatenates document `x` and document `y`. It is an associative operation having `empty` as a left and right unit.

`(<+>) :: Doc → Doc → Doc`

The document `(x <+> y)` concatenates document `x` and `y` with a `space` in between with identity `empty`.

`(<$$) :: Doc → Doc → Doc`

The document `(x <$$ y)` concatenates document `x` and `y` with a `line` in between with identity `empty`.

`(<$$+$$>) :: Doc → Doc → Doc`

The document `(x <$$+$$> y)` concatenates document `x` and `y` with a `blank line` in between with identity `empty`.

`(</>) :: Doc → Doc → Doc`

The document `(x </> y)` concatenates document `x` and `y` with a `softline` in between with identity `empty`. This effectively puts `x` and `y` either next to each other (with a `space` in between) or underneath each other.

`(<$$$>) :: Doc → Doc → Doc`

The document `(x <$$$> y)` concatenates document `x` and `y` with a `linebreak` in between with identity `empty`.

`(<///>) :: Doc → Doc → Doc`

The document `(x <///> y)` concatenates document `x` and `y` with a `softbreak` in between with identity `empty`. This effectively puts `x` and `y` either right next to each other or underneath each other.

```
(<$!$>) :: Doc → Doc → Doc
```

The document (`x <$!$> y`) concatenates document `x` and `y` with a `hardline` in between with identity `empty`. This effectively puts `x` and `y` underneath each other.

```
compose :: (Doc → Doc → Doc) → [Doc] → Doc
```

The document (`compose f xs`) concatenates all documents `xs` with function `f`. Function `f` should be like (`<+>`), (`$$`) and so on.

```
hsep :: [Doc] → Doc
```

The document (`hsep xs`) concatenates all documents `xs` horizontally with (`<+>`).

```
vsep :: [Doc] → Doc
```

The document (`vsep xs`) concatenates all documents `xs` vertically with (`$$`). If a group undoes the line breaks inserted by `vsep`, all documents are separated with a space.

```
someText = map text (words ("text to lay out"))
test     = text "some" <+> vsep someText
```

This is layed out as:

```
some text
to
lay
out
```

The `align` combinator can be used to align the documents under their first element:

```
test     = text "some" <+> align (vsep someText)
```

This is printed as:

```
some text
  to
  lay
  out
```

```
vsepBlank :: [Doc] → Doc
```

The document `vsep xs` concatenates all documents `xs` vertically with (`<$+$>`). If a group undoes the line breaks inserted by `vsepBlank`, all documents are separated with a space.

`fillSep :: [Doc] → Doc`

The document (`fillSep xs`) concatenates documents `xs` horizontally with (`</>`) as long as it fits the page, then inserts a `line` and continues doing that for all documents in `xs`. `fillSep xs = foldr (</>) empty xs`

`sep :: [Doc] → Doc`

The document (`sep xs`) concatenates all documents `xs` either horizontally with (`<+>`), if it fits the page, or vertically with (`$$$`). `sep xs = group (vsep xs)`

`hcat :: [Doc] → Doc`

The document (`hcat xs`) concatenates all documents `xs` horizontally with (`<>`).

`vcat :: [Doc] → Doc`

The document (`vcat xs`) concatenates all documents `xs` vertically with (`<$$$>`). If a `group` undoes the line breaks inserted by `vcat`, all documents are directly concatenated.

`fillCat :: [Doc] → Doc`

The document (`fillCat xs`) concatenates documents `xs` horizontally with (`<///>`) as long as it fits the page, then inserts a `linebreak` and continues doing that for all documents in `xs`. `fillCat xs = foldr (<///>) empty xs`

`cat :: [Doc] → Doc`

The document (`cat xs`) concatenates all documents `xs` either horizontally with (`<>`), if it fits the page, or vertically with (`<$$$>`). `cat xs = group (vcat xs)`

`punctuate :: Doc → [Doc] → [Doc]`

(`punctuate p xs`) concatenates all documents `xs` with document `p` except for the last document.

```
someText = map text ["words","in","a","tuple"]
test      = parens (align (cat (punctuate comma someText)))
```

This is layed out on a page width of 20 as:

```
(words,in,a,tuple)
```

But when the page width is 15, it is layed out as:

```
(words,
 in,
 a,
 tuple)
```

(If you want put the commas in front of their elements instead of at the end, you should use `tupled` or, in general, `encloseSep`.)

```
encloseSep :: Doc → Doc → Doc → [Doc] → Doc
```

The document (`encloseSep l r s xs`) concatenates the documents `xs` separated by `s` and encloses the resulting document by `l` and `r`. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All separators are put in front of the elements.

For example, the combinator `list` can be defined with `encloseSep`:

```
list xs = encloseSep lbracket rbracket comma xs
test    = text "list" <+> (list (map int [10,200,3000]))
```

Which is layed out with a page width of 20 as:

```
list [10,200,3000]
```

But when the page width is 15, it is layed out as:

```
list [10
      ,200
      ,3000]
```

```
encloseSepSpaced :: Doc → Doc → Doc → [Doc] → Doc
```

The document (`encloseSepSpaced l r s xs`) concatenates the documents `xs` separated by `s` and encloses the resulting document by `l` and `r`. In addition, after each occurrence of `s`, after `l`, and before `r`, a `space` is inserted. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All separators are put in front of the elements.

```
hEncloseSep :: Doc → Doc → Doc → [Doc] → Doc
```

The document (`hEncloseSep l r s xs`) concatenates the documents `xs` separated by `s` and encloses the resulting document by `l` and `r`.

The documents are rendered horizontally.

```
fillEncloseSep :: Doc → Doc → Doc → [Doc] → Doc
```

The document (`fillEncloseSep l r s xs`) concatenates the documents `xs` separated by `s` and encloses the resulting document by `l` and `r`.

The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All separators are put in front of the elements.

```
fillEncloseSepSpaced :: Doc → Doc → Doc → [Doc] → Doc
```

The document (`fillEncloseSepSpaced l r s xs`) concatenates the documents `xs` separated by `s` and encloses the resulting document by `l` and `r`. In addition, after each occurrence of `s`, after `l`, and before `r`, a space is inserted.

The documents are rendered horizontally if that fits the page. Otherwise, they are aligned vertically. All separators are put in front of the elements.

`list :: [Doc] → Doc`

The document (`list xs`) comma separates the documents `xs` and encloses them in square brackets. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma separators are put in front of the elements.

`listSpaced :: [Doc] → Doc`

Spaced version of `list`

`set :: [Doc] → Doc`

The document (`set xs`) comma separates the documents `xs` and encloses them in braces. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma separators are put in front of the elements.

`setSpaced :: [Doc] → Doc`

Spaced version of `set`

`tupled :: [Doc] → Doc`

The document (`tupled xs`) comma separates the documents `xs` and encloses them in parenthesis. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma separators are put in front of the elements.

`tupledSpaced :: [Doc] → Doc`

Spaced version of `tupled`

`semiBraces :: [Doc] → Doc`

The document (`semiBraces xs`) separates the documents `xs` with semi colons and encloses them in braces. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All semi colons are put in front of the elements.

`semiBracesSpaced :: [Doc] → Doc`

Spaced version of `semiBraces`

`enclose :: Doc → Doc → Doc → Doc`

The document (`enclose l r x`) encloses document `x` between documents `l` and `r` using (`<>`). `enclose l r x = l <> x <> r`

`squotes :: Doc → Doc`

Document (`squotes x`) encloses document `x` with single quotes `"'`".

`dquotes :: Doc → Doc`

Document (`dquotes x`) encloses document `x` with double quotes.

`bquotes :: Doc → Doc`

Document (`bquotes x`) encloses document `x` with back quotes `"`"`.

`parens :: Doc → Doc`

Document (`parens x`) encloses document `x` in parenthesis, `"(" and ")"`.

`parensIf :: Bool → Doc → Doc`

Document (`parensIf x`) encloses document `x` in parenthesis, `"(" and ")"`, iff the condition is true.

`angles :: Doc → Doc`

Document (`angles x`) encloses document `x` in angles, `"<" and ">"`.

`braces :: Doc → Doc`

Document (`braces x`) encloses document `x` in braces, `"{" and "}"`.

`brackets :: Doc → Doc`

Document (`brackets x`) encloses document `x` in square brackets, `"[" and "]"`.

`char :: Char → Doc`

The document (`char c`) contains the literal character `c`. The character should not be a newline (`\n`), the function `line` should be used for line breaks.

`string :: String → Doc`

The document (`string s`) concatenates all characters in `s` using `line` for newline characters and `char` for all other characters. It is used instead of `text` whenever the text contains newline characters.

`int :: Int → Doc`

The document (`int i`) shows the literal integer `i` using `text`.

`float :: Float → Doc`

The document (`float f`) shows the literal float `f` using `text`.

`lparen :: Doc`

The document `lparen` contains a left parenthesis, `"("`.

`rparen :: Doc`

The document `rparen` contains a right parenthesis, `)`.

`langle` :: Doc

The document `langle` contains a left angle, `<`.

`rangle` :: Doc

The document `rangle` contains a right angle, `>`.

`lbrace` :: Doc

The document `lbrace` contains a left brace, `{`.

`rbrace` :: Doc

The document `rbrace` contains a right brace, `}`.

`lbracket` :: Doc

The document `lbracket` contains a left square bracket, `[`.

`rbracket` :: Doc

The document `rbracket` contains a right square bracket, `]`.

`squote` :: Doc

The document `squote` contains a single quote, `'`.

`dquote` :: Doc

The document `dquote` contains a double quote.

`semi` :: Doc

The document `semi` contains a semi colon, `;`.

`colon` :: Doc

The document `colon` contains a colon, `:`.

`comma` :: Doc

The document `comma` contains a comma, `,`.

`space` :: Doc

The document `space` contains a single space, .

`x <+> y` = `x <> space <> y`

`dot` :: Doc

The document `dot` contains a single dot, `.`.

`backslash :: Doc`

The document `backslash` contains a back slash, `"\"`.

`equals :: Doc`

The document `equals` contains an equal sign, `"="`.

`larrow :: Doc`

The document `larrow` contains a left arrow sign, `"<-"`.

`rarrow :: Doc`

The document `rarrow` contains a right arrow sign, `"->"`.

`doubleArrow :: Doc`

The document `doubleArrow` contains an double arrow sign, `"=>"`.

`doubleColon :: Doc`

The document `doubleColon` contains a double colon sign, `"::"`.

`bar :: Doc`

The document `bar` contains a vertical bar sign, `"|"`.

`at :: Doc`

The document `at` contains an at sign, `"@"`.

`tilde :: Doc`

The document `tilde` contains a tilde sign, `"~"`.

`fill :: Int -> Doc -> Doc`

The document `(fill i d)` renders document `d`. It than appends spaces until the width is equal to `i`. If the width of `d` is already larger, nothing is appended. This combinator is quite useful in practice to output a list of bindings. The following example demonstrates this.

```
types = [("empty","Doc")
         ,("nest","Int -> Doc -> Doc")
         ,("linebreak","Doc")]
```

```
pptype (name,tp)
  = fill 6 (text name) <+> text ":@" <+> text tp
```

```
test = text "let" <+> align (vcap (map pptype types))
```

Which is layed out as:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak :: Doc
```

Note that `fill` is not guaranteed to be linear-time bounded since it has to compute the width of a document before pretty printing it

```
fillBreak :: Int → Doc → Doc
```

The document `(fillBreak i d)` first renders document `d`. It than appends spaces until the width is equal to `i`. If the width of `d` is already larger than `i`, the nesting level is increased by `i` and a line is appended. When we redefine `pType` in the previous example to use `fillBreak`, we get a useful variation of the previous output:

```
pType (name,tp)
    = fillBreak 6 (text name) <+> text ":@" <+> text tp
```

The output will now be:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak
        :: Doc
```

Note that `fillBreak` is not guaranteed to be linear-time bounded since it has to compute the width of a document before pretty printing it

```
bold :: Doc → Doc
```

The document `(bold d)` displays document `d` with bold text

```
faint :: Doc → Doc
```

The document `(faint d)` displays document `d` with faint text

```
blinkSlow :: Doc → Doc
```

The document `(blinkSlow d)` displays document `d` with slowly blinking text (rarely supported)

```
blinkRapid :: Doc → Doc
```

The document `(blinkRapid d)` displays document `d` with rapidly blinking text (rarely supported)

```
italic :: Doc → Doc
```

The document (`italic d`) displays document `d` with italicized text (rarely supported)

`underline :: Doc → Doc`

The document (`underline d`) displays document `d` with underlined text

`crossout :: Doc → Doc`

The document (`crossout d`) displays document `d` with crossed out text

`inverse :: Doc → Doc`

The document (`inverse d`) displays document `d` with inversed coloring, i.e. use text color of `d` as background color and background color of `d` as text color

`black :: Doc → Doc`

The document (`black d`) displays document `d` with black text color

`red :: Doc → Doc`

The document (`red d`) displays document `d` with red text color

`green :: Doc → Doc`

The document (`green d`) displays document `d` with green text color

`yellow :: Doc → Doc`

The document (`yellow d`) displays document `d` with yellow text color

`blue :: Doc → Doc`

The document (`blue d`) displays document `d` with blue text color

`magenta :: Doc → Doc`

The document (`magenta d`) displays document `d` with magenta text color

`cyan :: Doc → Doc`

The document (`cyan d`) displays document `d` with cyan text color

`white :: Doc → Doc`

The document (`white d`) displays document `d` with white text color

`bgBlack :: Doc → Doc`

The document (`bgBlack d`) displays document `d` with black background color

`bgRed :: Doc → Doc`

The document (`bgRed d`) displays document `d` with red background color

`bgGreen :: Doc → Doc`

The document (`bgGreen d`) displays document `d` with green background color

`bgYellow :: Doc → Doc`

The document (`bgYellow d`) displays document `d` with yellow background color

`bgBlue :: Doc → Doc`

The document (`bgBlue d`) displays document `d` with blue background color

`bgMagenta :: Doc → Doc`

The document (`bgMagenta d`) displays document `d` with magenta background color

`bgCyan :: Doc → Doc`

The document (`bgCyan d`) displays document `d` with cyan background color

`bgWhite :: Doc → Doc`

The document (`bgWhite d`) displays document `d` with white background color

`pretty :: Int → Doc → String`

(`pretty w d`) pretty prints document `d` with a page width of `w` characters

A.2.31 Library Profile

Preliminary library to support profiling.

Exported types:

`data ProcessInfo`

The data type for representing information about the state of a Curry process.

Exported constructors:

- `RunTime :: ProcessInfo`
`RunTime`
 - the run time in milliseconds
- `ElapsedTime :: ProcessInfo`
`ElapsedTime`
 - the elapsed time in milliseconds
- `Memory :: ProcessInfo`
`Memory`
 - the total memory in bytes

- `Code :: ProcessInfo`
`Code`
 - the size of the code area in bytes
- `Stack :: ProcessInfo`
`Stack`
 - the size of the local stack for recursive functions in bytes
- `Heap :: ProcessInfo`
`Heap`
 - the size of the heap to store term structures in bytes
- `Choices :: ProcessInfo`
`Choices`
 - the size of the choicepoint stack
- `GarbageCollections :: ProcessInfo`
`GarbageCollections`
 - the number of garbage collections performed

Exported functions:

`getProcessInfos :: IO [(ProcessInfo,Int)]`

Returns various informations about the current state of the Curry process. Note that the returned values are implementation dependent so that one should interpret them with care!

Note for kics2 users: Since GHC version 7.x, one has to set the run-time option `-T` when this operation is used. This can be done by the `kics2` command

```
:set rts -T
```

`garbageCollectorOff :: IO ()`

Turns off the garbage collector of the run-time system (if possible). This could be useful to get more precise data of memory usage.

`garbageCollectorOn :: IO ()`

Turns on the garbage collector of the run-time system (if possible).

`garbageCollect :: IO ()`

Invoke the garbage collector (if possible). This could be useful before run-time critical operations.

```
showMemInfo :: [(ProcessInfo,Int)] → String
```

Get a human readable version of the memory situation from the process infos.

```
printMemInfo :: IO ()
```

Print a human readable version of the current memory situation of the Curry process.

```
profileTime :: IO a → IO a
```

Print the time needed to execute a given IO action.

```
profileTimeNF :: a → IO ()
```

Evaluates the argument to normal form and print the time needed for this evaluation.

```
profileSpace :: IO a → IO a
```

Print the time and space needed to execute a given IO action. During the execution, the garbage collector is turned off to get the total space usage.

```
profileSpaceNF :: a → IO ()
```

Evaluates the argument to normal form and print the time and space needed for this evaluation. During the evaluation, the garbage collector is turned off to get the total space usage.

A.2.32 Library Prolog

A library defining a representation for Prolog programs together with a simple pretty printer. It does not cover all aspects of Prolog but might be useful for applications generating Prolog programs.

Exported types:

```
data PlClause
```

A Prolog clause is either a program clause consisting of a head and a body, or a directive or a query without a head.

Exported constructors:

- `PlClause :: String → [PlTerm] → [PlGoal] → PlClause`
- `PlDirective :: [PlGoal] → PlClause`
- `PlQuery :: [PlGoal] → PlClause`

```
data PlGoal
```

A Prolog goal is a literal, a negated goal, or a conditional.

Exported constructors:

- `PLit :: String → [PITerm] → PIGoal`
- `PNeg :: [PIGoal] → PIGoal`
- `PCond :: [PIGoal] → [PIGoal] → [PIGoal] → PIGoal`

`data PITerm`

A Prolog term is a variable, atom, number, or structure.

Exported constructors:

- `PVar :: String → PITerm`
- `PAtom :: String → PITerm`
- `PInt :: Int → PITerm`
- `PFloat :: Float → PITerm`
- `PStruct :: String → [PITerm] → PITerm`

Exported functions:

`pList :: [PITerm] → PITerm`

A Prolog list of Prolog terms.

`showPProg :: [PIClause] → String`

Shows a Prolog program in standard Prolog syntax.

`showPClause :: PIClause → String`

`showPGoals :: [PIGoal] → String`

`showPGoal :: PIGoal → String`

`showPITerm :: PITerm → String`

A.2.33 Library PropertyFile

A library to read and update files containing properties in the usual equational syntax, i.e., a property is defined by a line of the form `prop=value` where `prop` starts with a letter. All other lines (e.g., blank lines or lines starting with `#` are considered as comment lines and are ignored.

Exported functions:

`readPropertyFile :: String → IO [(String,String)]`

Reads a property file and returns the list of properties. Returns empty list if the property file does not exist.

`updatePropertyFile :: String → String → String → IO ()`

Update a property in a property file or add it, if it is not already there.

A.2.34 Library Read

Library with some functions for reading special tokens.

This library is included for backward compatibility. You should use the library `ReadNumeric` which provides a better interface for these functions.

Exported functions:

`readNat :: String → Int`

Read a natural number in a string. The string might contain leading blanks and the the number is read up to the first non-digit.

`readInt :: String → Int`

Read a (possibly negative) integer in a string. The string might contain leading blanks and the the integer is read up to the first non-digit.

`readHex :: String → Int`

Read a hexadecimal number in a string. The string might contain leading blanks and the the integer is read up to the first non-hexadecimal digit.

A.2.35 Library ReadNumeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

`readInt :: String → Maybe (Int,String)`

Read a (possibly negative) integer as a first token in a string. The string might contain leading blanks and the integer is read up to the first non-digit. If the string does not start with an integer token, `Nothing` is returned, otherwise the result is `Just (v, s)`, where `v` is the value of the integer and `s` is the remaining string without the integer token.

`readNat :: String → Maybe (Int,String)`

Read a natural number as a first token in a string. The string might contain leading blanks and the number is read up to the first non-digit. If the string does not start with a natural number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where `v` is the value of the number and `s` is the remaining string without the number token.

```
readHex :: String → Maybe (Int,String)
```

Read a hexadecimal number as a first token in a string. The string might contain leading blanks and the number is read up to the first non-hexadecimal digit. If the string does not start with a hexadecimal number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where `v` is the value of the number and `s` is the remaining string without the number token.

```
readOct :: String → Maybe (Int,String)
```

Read an octal number as a first token in a string. The string might contain leading blanks and the number is read up to the first non-octal digit. If the string does not start with an octal number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where `v` is the value of the number and `s` is the remaining string without the number token.

A.2.36 Library ReadShowTerm

Library for converting ground terms to strings and vice versa.

Exported functions:

```
showTerm :: a → String
```

Transforms a ground(!) term into a string representation in standard prefix notation. Thus, `showTerm` suspends until its argument is ground. This function is similar to the prelude function `show` but can read the string back with `readUnqualifiedTerm` (provided that the constructor names are unique without the module qualifier).

```
showQTerm :: a → String
```

Transforms a ground(!) term into a string representation in standard prefix notation. Thus, `showTerm` suspends until its argument is ground. Note that this function differs from the prelude function `show` since it prefixes constructors with their module name in order to read them back with `readQTerm`.

```
readsUnqualifiedTerm :: [String] → String → [(a,String)]
```

Transform a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The first argument is a non-empty list of module qualifiers that are tried to prefix the constructor in the string in order to get the qualified constructors (that must be defined in the current program!). In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

`readUnqualifiedTerm :: [String] → String → a`

Transforms a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The first argument is a non-empty list of module qualifiers that are tried to prefix the constructor in the string in order to get the qualified constructors (that must be defined in the current program!).

Example: `readUnqualifiedTerm ["Prelude"] "Just 3"` evaluates to `(Just 3)`

`readsTerm :: String → [(a,String)]`

For backward compatibility. Should not be used since their use can be problematic in case of constructors with identical names in different modules.

`readTerm :: String → a`

For backward compatibility. Should not be used since their use can be problematic in case of constructors with identical names in different modules.

`readsQTerm :: String → [(a,String)]`

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term. In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

`readQTerm :: String → a`

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term.

`readQTermFile :: String → IO a`

Reads a file containing a string representation of a term in standard prefix notation and returns the corresponding data term.

`readQTermListFile :: String → IO [a]`

Reads a file containing lines with string representations of terms of the same type and returns the corresponding list of data terms.

`writeQTermFile :: String → a → IO ()`

Writes a ground term into a file in standard prefix notation.

`writeQTermListFile :: String → [a] → IO ()`

Writes a list of ground terms into a file. Each term is written into a separate line which might be useful to modify the file with a standard text editor.

A.2.37 Library SearchTree

This library defines a representation of a search space as a tree and various search strategies on this tree. This module implements **strong encapsulation** as discussed in [this paper](#)

Exported types:

`type Strategy a = SearchTree a → ValueSequence a`

A search strategy maps a search tree into some sequence of values. Using the abstract type of sequence of values (rather than list of values) enables the use of search strategies for encapsulated search with search trees (strong encapsulation) as well as with set functions (weak encapsulation).

`data SearchTree`

A search tree is a value, a failure, or a choice between two search trees.

Exported constructors:

- `Value :: a → SearchTree a`
- `Fail :: Int → SearchTree a`
- `Or :: (SearchTree a) → (SearchTree a) → SearchTree a`

Exported functions:

`getSearchTree :: a → IO (SearchTree a)`

Returns the search tree for some expression.

`someSearchTree :: a → SearchTree a`

Internal operation to return the search tree for some expression. Note that this operation is not purely declarative since the ordering in the resulting search tree depends on the ordering of the program rules.

`isDefined :: a → Bool`

Returns True iff the argument is defined, i.e., has a value.

`showSearchTree :: SearchTree a → String`

Shows the search tree as an intended line structure

`searchTreeSize :: SearchTree a → (Int,Int,Int)`

Return the size (number of Value/Fail/Or nodes) of the search tree

`dfsStrategy :: SearchTree a → ValueSequence a`

Depth-first search strategy.

`bfsStrategy :: SearchTree a → ValueSequence a`

Breadth-first search strategy.

`idsStrategy :: SearchTree a → ValueSequence a`

Iterative-deepening search strategy.

`idsStrategyWith :: Int → (Int → Int) → SearchTree a → ValueSequence a`

Parameterized iterative-deepening search strategy. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`diagStrategy :: SearchTree a → ValueSequence a`

Diagonalization search strategy.

`allValuesWith :: (SearchTree a → ValueSequence a) → SearchTree a → [a]`

Return all values in a search tree via some given search strategy.

`allValuesDFS :: SearchTree a → [a]`

Return all values in a search tree via depth-first search.

`allValuesBFS :: SearchTree a → [a]`

Return all values in a search tree via breadth-first search.

`allValuesIDS :: SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search.

`allValuesIDSwith :: Int → (Int → Int) → SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`allValuesDiag :: SearchTree a → [a]`

Return all values in a search tree via diagonalization search strategy.

`getAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO [a]`

Gets all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`printAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO ()`

Prints all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all printed values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`printValuesWith :: (SearchTree a → ValueSequence a) → a → IO ()`

Prints the values of an expression w.r.t. a search strategy on demand by the user. Thus, the user must type `<enter></enter>` before another value is computed and printed. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all printed values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`someValue :: a → a`

Returns some value for an expression.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

`someValueWith :: (SearchTree a → ValueSequence a) → a → a`

Returns some value for an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

A.2.38 Library SetFunctions

This module contains an implementation of set functions. The general idea of set functions is described in:

S. Antoy, M. Hanus: Set Functions for Functional Logic Programming Proc. 11th International Conference on Principles and Practice of Declarative Programming (PPDP'09), pp. 73-82, ACM Press, 2009

Intuition: If `f` is an n -ary function, then `(setn f)` is a set-valued function that collects all non-determinism caused by `f` (but not the non-determinism caused by evaluating arguments!) in a set. Thus, `(setn f a1 ... an)` returns the set of all values of `(f b1 ... bn)` where `b1,...,bn` are values of the arguments `a1,...,an` (i.e., the arguments are evaluated "outside" this capsule so that the non-determinism caused by evaluating these arguments is not captured in this capsule but yields several results for `(setn...)`). Similarly, logical variables occurring in `a1,...,an` are not bound inside this capsule.

The set of values returned by a set function is represented by an abstract type `Values` on which several operations are defined in this module. Actually, it is a multiset of values, i.e., duplicates are not removed.

The handling of failures and nested occurrences of set functions is not specified in the previous paper. Thus, a detailed description of the semantics of set functions as implemented in this library can be found in the paper

J. Christiansen, M. Hanus, F. Reck, D. Seidel: A Semantics for Weakly Encapsulated Search in Functional Logic Programs Proc. 15th International Conference on Principles and Practice of Declarative Programming (PPDP'13), pp. 49-60, ACM Press, 2013

Exported types:

`data Values`

Abstract type representing multisets of values.

Exported constructors:

Exported functions:

`set0 :: a → Values a`

Combinator to transform a 0-ary function into a corresponding set function.

`set0With :: (SearchTree a → ValueSequence a) → a → Values a`

Combinator to transform a 0-ary function into a corresponding set function that uses a given strategy to compute its values.

`set1 :: (a → b) → a → Values b`

Combinator to transform a unary function into a corresponding set function.

`set1With :: (SearchTree a → ValueSequence a) → (b → a) → b → Values a`

Combinator to transform a unary function into a corresponding set function that uses a given strategy to compute its values.

`set2 :: (a → b → c) → a → b → Values c`

Combinator to transform a binary function into a corresponding set function.

`set2With :: (SearchTree a → ValueSequence a) → (b → c → a) → b → c → Values a`

Combinator to transform a binary function into a corresponding set function that uses a given strategy to compute its values.

`set3 :: (a → b → c → d) → a → b → c → Values d`

Combinator to transform a function of arity 3 into a corresponding set function.

`set3With :: (SearchTree a → ValueSequence a) → (b → c → d → a) → b → c → d → Values a`

Combinator to transform a function of arity 3 into a corresponding set function that uses a given strategy to compute its values.

`set4 :: (a → b → c → d → e) → a → b → c → d → Values e`

Combinator to transform a function of arity 4 into a corresponding set function.

`set4With :: (SearchTree a → ValueSequence a) → (b → c → d → e → a) → b → c → d → e → Values a`

Combinator to transform a function of arity 4 into a corresponding set function that uses a given strategy to compute its values.

```
set5 :: (a → b → c → d → e → f) → a → b → c → d → e → Values f
```

Combinator to transform a function of arity 5 into a corresponding set function.

```
set5With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → a) → b  
→ c → d → e → f → Values a
```

Combinator to transform a function of arity 5 into a corresponding set function that uses a given strategy to compute its values.

```
set6 :: (a → b → c → d → e → f → g) → a → b → c → d → e → f → Values  
g
```

Combinator to transform a function of arity 6 into a corresponding set function.

```
set6With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → g → a)  
→ b → c → d → e → f → g → Values a
```

Combinator to transform a function of arity 6 into a corresponding set function that uses a given strategy to compute its values.

```
set7 :: (a → b → c → d → e → f → g → h) → a → b → c → d → e → f → g  
→ Values h
```

Combinator to transform a function of arity 7 into a corresponding set function.

```
set7With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → g → h  
→ a) → b → c → d → e → f → g → h → Values a
```

Combinator to transform a function of arity 7 into a corresponding set function that uses a given strategy to compute its values.

```
isEmpty :: Values a → Bool
```

Is a multiset of values empty?

```
notEmpty :: Values a → Bool
```

Is a multiset of values not empty?

```
valueOf :: a → Values a → Bool
```

Is some value an element of a multiset of values?

```
choose :: Values a → (a, Values a)
```

Chooses (non-deterministically) some value in a multiset of values and returns the chosen value and the remaining multiset of values. Thus, if we consider the operation `chooseValue` by

`chooseValue x = fst (choose x)`

then `(set1 chooseValue)` is the identity on value sets, i.e., `(set1 chooseValue s)` contains the same elements as the value set `s`.

`chooseValue :: Values a → a`

Chooses (non-deterministically) some value in a multiset of values and returns the chosen value. Thus, `(set1 chooseValue)` is the identity on value sets, i.e., `(set1 chooseValue s)` contains the same elements as the value set `s`.

`select :: Values a → (a, Values a)`

Selects (indeterministically) some value in a multiset of values and returns the selected value and the remaining multiset of values. Thus, `select` has always at most one value. It fails if the value set is empty.

NOTE: The usage of this operation is only safe (i.e., does not destroy completeness) if all values in the argument set are identical.

`selectValue :: Values a → a`

Selects (indeterministically) some value in a multiset of values and returns the selected value. Thus, `selectValue` has always at most one value. It fails if the value set is empty.

NOTE: The usage of this operation is only safe (i.e., does not destroy completeness) if all values in the argument set are identical.

`mapValues :: (a → b) → Values a → Values b`

Accumulates all elements of a multiset of values by applying a binary operation. This is similarly to fold on lists, but the binary operation must be **commutative** so that the result is independent of the order of applying this operation to all elements in the multiset.

`foldValues :: (a → a → a) → a → Values a → a`

Accumulates all elements of a multiset of values by applying a binary operation. This is similarly to fold on lists, but the binary operation must be **commutative** so that the result is independent of the order of applying this operation to all elements in the multiset.

`minValue :: (a → a → Bool) → Values a → a`

Returns the minimal element of a non-empty multiset of values with respect to a given total ordering on the elements.

`maxValue :: (a → a → Bool) → Values a → a`

Returns the maximal element of a non-empty multiset of value with respect to a given total ordering on the elements.

`values2list :: Values a → IO [a]`

Puts all elements of a multiset of values in a list. Since the order of the elements in the list might depend on the time of the computation, this operation is an I/O action.

`printValues :: Values a → IO ()`

Prints all elements of a multiset of values.

`sortValues :: Values a → [a]`

Transforms a multiset of values into a list sorted by the standard term ordering. As a consequence, the multiset of values is completely evaluated.

`sortValuesBy :: (a → a → Bool) → Values a → [a]`

Transforms a multiset of values into a list sorted by a given ordering on the values. As a consequence, the multiset of values is completely evaluated. In order to ensure that the result of this operation is independent of the evaluation order, the given ordering must be a total order.

A.2.39 Library Socket

Library to support network programming with sockets. In standard applications, the server side uses the operations `listenOn` and `socketAccept` to provide some service on a socket, and the client side uses the operation `connectToSocket` to request a service.

Exported types:

`data Socket`

The abstract type of sockets.

Exported constructors:

Exported functions:

`listenOn :: Int → IO Socket`

Creates a server side socket bound to a given port number.

`listenOnFresh :: IO (Int,Socket)`

Creates a server side socket bound to a free port. The port number and the socket is returned.

`socketAccept :: Socket → IO (String,Handle)`

Returns a connection of a client to a socket. The connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client. The handle is both readable and writable.

`waitForSocketAccept :: Socket → Int → IO (Maybe (String,Handle))`

Waits until a connection of a client to a socket is available. If no connection is available within the time limit, it returns `Nothing`, otherwise the connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client.

`sClose :: Socket → IO ()`

Closes a server socket.

`connectToSocket :: String → Int → IO Handle`

Creates a new connection to a Unix socket.

A.2.40 Library System

Library to access parts of the system environment.

Exported functions:

`getCPUTime :: IO Int`

Returns the current cpu time of the process in milliseconds.

`getElapsedTime :: IO Int`

Returns the current elapsed time of the process in milliseconds. This operation is not supported in KiCS2 (there it always returns 0), but only included for compatibility reasons.

`getArgs :: IO [String]`

Returns the list of the program's command line arguments. The program name is not included.

`getEnviron :: String → IO String`

Returns the value of an environment variable. The empty string is returned for undefined environment variables.

`setEnviron :: String → String → IO ()`

Set an environment variable to a value. The new value will be passed to subsequent shell commands (see `system`) and visible to subsequent calls to `getEnviron` (but it is not visible in the environment of the process that started the program execution).

`unsetEnviron :: String → IO ()`

Removes an environment variable that has been set by `setEnviron`.

`getHostname :: IO String`

Returns the hostname of the machine running this process.

`getPID :: IO Int`

Returns the process identifier of the current Curry process.

`getProgName :: IO String`

Returns the name of the current program, i.e., the name of the main module currently executed.

`system :: String → IO Int`

Executes a shell command and return with the exit code of the command. An exit status of zero means successful execution.

`exitWith :: Int → IO a`

Terminates the execution of the current Curry program and returns the exit code given by the argument. An exit code of zero means successful execution.

`sleep :: Int → IO ()`

The evaluation of the action (`sleep n`) puts the Curry process asleep for `n` seconds.

`isPosix :: Bool`

Is the underlying operating system a POSIX system (unix, MacOS)?

`isWindows :: Bool`

Is the underlying operating system a Windows system?

A.2.41 Library Time

Library for handling date and time information.

Exported types:

`data ClockTime`

`ClockTime` represents a clock time in some internal representation.

Exported constructors:

`data CalendarTime`

A calendar time is presented in the following form: (`CalendarTime year month day hour minute second timezone`) where `timezone` is an integer representing the timezone as a difference to UTC time in seconds.

Exported constructors:

- `CalendarTime :: Int → Int → Int → Int → Int → Int → Int → Int → CalendarTime`

Exported functions:

`ctYear :: CalendarTime → Int`

The year of a calendar time.

`ctMonth :: CalendarTime → Int`

The month of a calendar time.

`ctDay :: CalendarTime → Int`

The day of a calendar time.

`ctHour :: CalendarTime → Int`

The hour of a calendar time.

`ctMin :: CalendarTime → Int`

The minute of a calendar time.

`ctSec :: CalendarTime → Int`

The second of a calendar time.

`ctTZ :: CalendarTime → Int`

The time zone of a calendar time. The value of the time zone is the difference to UTC time in seconds.

`getClockTime :: IO ClockTime`

Returns the current clock time.

`getLocalTime :: IO CalendarTime`

Returns the local calendar time.

`clockTimeToInt :: ClockTime → Int`

Transforms a clock time into a unique integer. It is ensured that clock times that differs in at least one second are mapped into different integers.

`toCalendarTime :: ClockTime → IO CalendarTime`

Transforms a clock time into a calendar time according to the local time (if possible). Since the result depends on the local environment, it is an I/O operation.

`toUTCTime :: ClockTime → CalendarTime`

Transforms a clock time into a standard UTC calendar time. Thus, this operation is independent on the local time.

`toClockTime :: CalendarTime → ClockTime`

Transforms a calendar time (interpreted as UTC time) into a clock time.

`calendarTimeToString :: CalendarTime → String`

Transforms a calendar time into a readable form.

`toDayString :: CalendarTime → String`

Transforms a calendar time into a string containing the day, e.g., "September 23, 2006".

`toTimeString :: CalendarTime → String`

Transforms a calendar time into a string containing the time.

`addSeconds :: Int → ClockTime → ClockTime`

Adds seconds to a given time.

`addMinutes :: Int → ClockTime → ClockTime`

Adds minutes to a given time.

`addHours :: Int → ClockTime → ClockTime`

Adds hours to a given time.

`addDays :: Int → ClockTime → ClockTime`

Adds days to a given time.

`addMonths :: Int → ClockTime → ClockTime`

Adds months to a given time.

`addYears :: Int → ClockTime → ClockTime`

Adds years to a given time.

`daysOfMonth :: Int → Int → Int`

Gets the days of a month in a year.

`validDate :: Int → Int → Int → Bool`

Is a date consisting of year/month/day valid?

`compareDate :: CalendarTime → CalendarTime → Ordering`

Compares two dates (don't use it, just for backward compatibility!).

`compareCalendarTime :: CalendarTime → CalendarTime → Ordering`

Compares two calendar times.

`compareClockTime :: ClockTime → ClockTime → Ordering`

Compares two clock times.

A.2.42 Library Unsafe

Library containing unsafe operations. These operations should be carefully used (e.g., for testing or debugging). These operations should not be used in application programs!

Exported functions:

`unsafePerformIO :: IO a → a`

Performs and hides an I/O action in a computation (use with care!).

`trace :: String → a → a`

Prints the first argument as a side effect and behaves as identity on the second argument.

A.2.43 Library UnsafeSearchTree

This library defines a representation of a search space as a tree and various search strategies on this tree. This module implements **strong encapsulation** as discussed in [this paper](#)

Warning: In contrast to the SearchTree Module, free variables that are not bound in the encapsulated expression remain free! This may lead to non-determinism if such an escaped variable is bound later via pattern matching.

Exported types:

`type Strategy a = SearchTree a → ValueSequence a`

`data SearchTree`

A search tree is a value, a failure, or a choice between two search trees.

Exported constructors:

- `Value :: a → SearchTree a`
- `Fail :: Int → SearchTree a`
- `Or :: (SearchTree a) → (SearchTree a) → SearchTree a`

Exported functions:

`isVar :: a → Bool`

Tests whether the argument is a free variable This function is only meaningful when applied to a part of a result of an encapsulated expression if the argument stems from a Value node of a SearchTree

`identicalVars :: a → a → Bool`

Tests whether both arguments are identical free variables. This function is only meaningful when applied to parts of a result of an encapsulated expression if the argument stems from a `Value` node of a `SearchTree`

`varId :: a → Int`

Returns the unique identifier of a free variable, if the argument was not a free variable, otherwise an error is raised. This function is only meaningful when applied to a part of a result of an encapsulated expression if the argument stems from a `Value` node of a `SearchTree`

`getSearchTree :: a → IO (SearchTree a)`

Returns the search tree for some expression.

`someSearchTree :: a → SearchTree a`

Internal operation to return the search tree for some expression. Note that this operation is not purely declarative since the ordering in the resulting search tree depends on the ordering of the program rules.

`isDefined :: a → Bool`

Returns `True` iff the argument is is defined, i.e., has a value.

`showSearchTree :: SearchTree a → String`

Shows the search tree as an intended line structure

`searchTreeSize :: SearchTree a → (Int,Int,Int)`

Return the size (number of `Value`/`Fail`/`Or` nodes) of the search tree

`allValuesDFS :: SearchTree a → [a]`

Return all values in a search tree via depth-first search

`dfsStrategy :: SearchTree a → ValueSequence a`

`allValuesBFS :: SearchTree a → [a]`

Return all values in a search tree via breadth-first search

`bfsStrategy :: SearchTree a → ValueSequence a`

`allValuesIDS :: SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search.

`idsStrategy :: SearchTree a → ValueSequence a`

`allValuesIDsWith :: Int → (Int → Int) → SearchTree a → [a]`

Return the list of all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`idsStrategyWith :: Int → (Int → Int) → SearchTree a → ValueSequence a`

Return all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`getAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO [a]`

Gets all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

`someValue :: a → a`

Returns some value for an expression.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

`someValueWith :: (SearchTree a → ValueSequence a) → a → a`

Returns some value for an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

A.2.44 Library ValueSequence

This library defines a data structure for sequence of values. It is used in search trees (module `SearchTree`) as well as in set functions (module `SetFunctions`). Using sequence of values (rather than standard lists of values) is necessary to get the behavior of set functions w.r.t. finite failures right, as described in the paper

J. Christiansen, M. Hanus, F. Reck, D. Seidel: A Semantics for Weakly Encapsulated Search in Functional Logic Programs Proc. 15th International Conference on Principles and Practice of Declarative Programming (PPDP'13), pp. 49-60, ACM Press, 2013

Exported types:

`data ValueSequence`

A value sequence is an abstract sequence of values. It also contains failure elements in order to implement the semantics of set functions w.r.t. failures in the intended manner.

Exported constructors:

Exported functions:

`emptyVS :: ValueSequence a`

An empty sequence of values.

`addVS :: a → ValueSequence a → ValueSequence a`

Adds a value to a sequence of values.

`failVS :: Int → ValueSequence a`

Adds a failure to a sequence of values. The argument is the encapsulation level of the failure.

`(|++|) :: ValueSequence a → ValueSequence a → ValueSequence a`

Concatenates two sequences of values.

`vsToList :: ValueSequence a → [a]`

Transforms a sequence of values into a list of values.

A.3 Data Structures and Algorithms

A.3.1 Library Array

Implementation of Arrays with Braun Trees. Conceptually, Braun trees are always infinite. Consequently, there is no test on emptiness.

Exported types:

`data Array`

Exported constructors:

Exported functions:

`emptyErrorArray :: Array a`

Creates an empty array which generates errors for non-initialized indexes.

`emptyDefaultArray :: (Int → a) → Array a`

Creates an empty array, call given function for non-initialized indexes.

`(//) :: Array a → [(Int,a)] → Array a`

Inserts a list of entries into an array.

`update :: Array a → Int → a → Array a`

Inserts a new entry into an array.

`applyAt :: Array a → Int → (a → a) → Array a`

Applies a function to an element.

`(!) :: Array a → Int → a`

Yields the value at a given position.

`listToDefaultArray :: (Int → a) → [a] → Array a`

Creates a default array from a list of entries.

`listToErrorArray :: [a] → Array a`

Creates an error array from a list of entries.

`combine :: (a → b → c) → Array a → Array b → Array c`

combine two arbitrary arrays

`combineSimilar :: (a → a → a) → Array a → Array a → Array a`

the combination of two arrays with identical default function and a combinator which is neutral in the default can be implemented much more efficient

A.3.2 Library Dequeue

An implementation of double-ended queues supporting access at both ends in constant amortized time.

Exported types:

`data Queue`

The datatype of a queue.

Exported constructors:

Exported functions:

`empty :: Queue a`

The empty queue.

`cons :: a → Queue a → Queue a`

Inserts an element at the front of the queue.

`snoc :: a → Queue a → Queue a`

Inserts an element at the end of the queue.

`isEmpty :: Queue a → Bool`

Is the queue empty?

`deqLength :: Queue a → Int`

Returns the number of elements in the queue.

`deqHead :: Queue a → a`

The first element of the queue.

`deqTail :: Queue a → Queue a`

Removes an element at the front of the queue.

`deqLast :: Queue a → a`

The last element of the queue.

`deqInit :: Queue a → Queue a`

Removes an element at the end of the queue.

`deqReverse :: Queue a → Queue a`

Reverses a double ended queue.

`rotate :: Queue a → Queue a`

Moves the first element to the end of the queue.

`matchHead :: Queue a → Maybe (a, Queue a)`

Matches the front of a queue. `matchHead q` is equivalent to `if isEmpty q then Nothing else Just (deqHead q, deqTail q)` but more efficient.

`matchLast :: Queue a → Maybe (a, Queue a)`

Matches the end of a queue. `matchLast q` is equivalent to `if isEmpty q then Nothing else Just (deqLast q, deqInit q)` but more efficient.

`listToDeq :: [a] → Queue a`

Transforms a list to a double ended queue.

`deqToList :: Queue a → [a]`

Transforms a double ended queue to a list.

A.3.3 Library FiniteMap

A finite map is an efficient purely functional data structure to store a mapping from keys to values. In order to store the mapping efficiently, an irreflexive(!) order predicate has to be given, i.e., the order predicate `le` should not satisfy `(le x x)` for some key `x`.

Example: To store a mapping from `Int -> String`, the finite map needs a Boolean predicate like `(<)`. This version was ported from a corresponding Haskell library

Exported types:

`data FM`

Exported constructors:

Exported functions:

`emptyFM :: (a -> a -> Bool) -> FM a b`

The empty finite map.

`unitFM :: (a -> a -> Bool) -> a -> b -> FM a b`

Construct a finite map with only a single element.

`listToFM :: (a -> a -> Bool) -> [(a,b)] -> FM a b`

Builds a finite map from given list of tuples (key,element). For multiple occurrences of key, the last corresponding element of the list is taken.

`addToFM :: FM a b -> a -> b -> FM a b`

Throws away any previous binding and stores the new one given.

`addListToFM :: FM a b -> [(a,b)] -> FM a b`

Throws away any previous bindings and stores the new ones given. The items are added starting with the first one in the list

`addToFM_C :: (a -> a -> a) -> FM b a -> b -> a -> FM b a`

Instead of throwing away the old binding, `addToFM_C` combines the new element with the old one.

`addListToFM_C :: (a -> a -> a) -> FM b a -> [(b,a)] -> FM b a`

Combine with a list of tuples (key,element), cf. `addToFM_C`

`delFromFM :: FM a b -> a -> FM a b`

Deletes key from finite map. Deletion doesn't complain if you try to delete something which isn't there

`dellistFromFM :: FM a b → [a] → FM a b`

Deletes a list of keys from finite map. Deletion doesn't complain if you try to delete something which isn't there

`updFM :: FM a b → a → (b → b) → FM a b`

Applies a function to element bound to given key.

`splitFM :: FM a b → a → Maybe (FM a b, (a,b))`

Combines delFrom and lookup.

`plusFM :: FM a b → FM a b → FM a b`

Efficiently add key/element mappings of two maps into a single one. Bindings in right argument shadow those in the left

`plusFM_C :: (a → a → a) → FM b a → FM b a → FM b a`

Efficiently combine key/element mappings of two maps into a single one, cf. `addToFM_C`

`minusFM :: FM a b → FM a b → FM a b`

(`minusFM a1 a2`) deletes from `a1` any bindings which are bound in `a2`

`intersectFM :: FM a b → FM a b → FM a b`

Filters only those keys that are bound in both of the given maps. The elements will be taken from the second map.

`intersectFM_C :: (a → b → c) → FM d a → FM d b → FM d c`

Filters only those keys that are bound in both of the given maps and combines the elements as in `addToFM_C`.

`foldFM :: (a → b → c → c) → c → FM a b → c`

Folds finite map by given function.

`mapFM :: (a → b → c) → FM a b → FM a c`

Applies a given function on every element in the map.

`filterFM :: (a → b → Bool) → FM a b → FM a b`

Yields a new finite map with only those key/element pairs matching the given predicate.

`sizeFM :: FM a b → Int`

How many elements does given map contain?

`eqFM :: FM a b → FM a b → Bool`

Do two given maps contain the same key/element pairs?

`isEmptyFM :: FM a b → Bool`

Is the given finite map empty?

`elemFM :: a → FM a b → Bool`

Does given map contain given key?

`lookupFM :: FM a b → a → Maybe b`

Retrieves element bound to given key

`lookupWithDefaultFM :: FM a b → b → a → b`

Retrieves element bound to given key. If the element is not contained in map, return default value.

`keyOrder :: FM a b → a → a → Bool`

Retrieves the ordering on which the given finite map is built.

`minFM :: FM a b → Maybe (a,b)`

Retrieves the smallest key/element pair in the finite map according to the basic key ordering.

`maxFM :: FM a b → Maybe (a,b)`

Retrieves the greatest key/element pair in the finite map according to the basic key ordering.

`fmToList :: FM a b → [(a,b)]`

Builds a list of key/element pairs. The list is ordered by the initially given irreflexive order predicate on keys.

`keysFM :: FM a b → [a]`

Retrieves a list of keys contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`eltsFM :: FM a b → [b]`

Retrieves a list of elements contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`fmToListPreOrder :: FM a b → [(a,b)]`

Retrieves list of key/element pairs in preorder of the internal tree. Useful for lists that will be retransformed into a tree or to match any elements regardless of basic order.

`fmSortBy :: (a → a → Bool) → [a] → [a]`

Sorts a given list by inserting and retrieving from finite map. Duplicates are deleted.

```
showFM :: FM a b → String
```

Transforms a finite map into a string. For efficiency reasons, the tree structure is shown which is valid for reading only if one uses the same ordering predicate.

```
readFM :: (a → a → Bool) → String → FM a b
```

Transforms a string representation of a finite map into a finite map. One has to provide the same ordering predicate as used in the original finite map.

A.3.4 Library GraphInductive

Library for inductive graphs (part of a Haskell library by Martin Erwig).

In this library, graphs are composed and decomposed in an inductive way.

The key idea is as follows:

A graph is either *empty* or it consists of *node context* and a *graph g'* which are put together by a constructor (`:&`).

This constructor (`:&`), however, is not a constructor in the sense of abstract data type, but more basically a defined constructing function.

A *context* is a node together with the edges to and from this node into the nodes in the graph *g'*. For examples of how to use this library, cf. the module `GraphAlgorithms`.

Exported types:

```
type Node = Int
```

Nodes and edges themselves (in contrast to their labels) are coded as integers.

For both of them, there are variants as labeled, unlabeled and quasi unlabeled (labeled with `()`).

Unlabeled node

```
type LNode a = (Int,a)
```

Labeled node

```
type UNode = (Int,())
```

Quasi-unlabeled node

```
type Edge = (Int,Int)
```

Unlabeled edge

```
type LEdge a = (Int,Int,a)
```

Labeled edge

```
type UEdge = (Int,Int,())
```

Quasi-unlabeled edge

```
type Context a b = ([b,Int],Int,a,[b,Int])
```

The context of a node is the node itself (along with label) and its adjacent nodes. Thus, a context is a quadrupel, for node n it is of the form (edges to n , node n , n 's label, edges from n)

```
type MContext a b = Maybe ([b,Int],Int,a,[b,Int])
```

maybe context

```
type Context' a b = ([b,Int],a,[b,Int])
```

context with edges and node label only, without the node identifier itself

```
type UContext = ([Int],Int,[Int])
```

Unlabeled context.

```
type GDecomp a b = (([b,Int],Int,a,[b,Int]),Graph a b)
```

A graph decomposition is a context for a node n and the remaining graph without that node.

```
type Decomp a b = (Maybe ([b,Int],Int,a,[b,Int]),Graph a b)
```

a decomposition with a maybe context

```
type UDecomp a = (Maybe ([Int],Int,[Int]),a)
```

Unlabeled decomposition.

```
type Path = [Int]
```

Unlabeled path

```
type LPath a = [(Int,a)]
```

Labeled path

```
type UPath = [(Int,())]
```

Quasi-unlabeled path

```
type UGr = Graph () ()
```

a graph without any labels

```
data Graph
```

The type variables of `Graph` are *nodeLabel* and *edgeLabel*. The internal representation of `Graph` is hidden.

Exported constructors:

Exported functions:

`(:&) :: [(a,Int)],Int,b,[(a,Int))] → Graph b a → Graph b a`

`(:&)` takes a node-context and a Graph and yields a new graph.

The according key idea is detailed at the beginning.

`nl` is the type of the node labels and `el` the edge labels.

Note that it is an error to induce a context for a node already contained in the graph.

`matchAny :: Graph a b → (((b,Int)],Int,a,[(b,Int)]),Graph a b)`

decompose a graph into the Context for an arbitrarily-chosen Node and the remaining Graph.

In order to use graphs as abstract data structures, we also need means to decompose a graph. This decomposition should work as much like pattern matching as possible. The normal matching is done by the function `matchAny`, which takes a graph and yields a graph decomposition.

According to the main idea, `matchAny . (:&)` should be an identity.

`empty :: Graph a b`

An empty Graph.

`mkGraph :: [(Int,a)] → [(Int,Int,b)] → Graph a b`

Create a Graph from the list of LNodes and LEdges.

`buildGr :: [(a,Int)],Int,b,[(a,Int)]] → Graph b a`

Build a Graph from a list of Contexts.

`mkUGraph :: [Int] → [(Int,Int)] → Graph () ()`

Build a quasi-unlabeled Graph from the list of Nodes and Edges.

`insNode :: (Int,a) → Graph a b → Graph a b`

Insert a LNode into the Graph.

`insEdge :: (Int,Int,a) → Graph b a → Graph b a`

Insert a LEdge into the Graph.

`delNode :: Int → Graph a b → Graph a b`

Remove a Node from the Graph.

`delEdge :: (Int,Int) → Graph a b → Graph a b`

Remove an Edge from the Graph.

`insNodes :: [(Int,a)] → Graph a b → Graph a b`

Insert multiple LNodes into the Graph.

```
insEdges :: [(Int,Int,a)] → Graph b a → Graph b a
```

Insert multiple LEdges into the Graph.

```
delNodes :: [Int] → Graph a b → Graph a b
```

Remove multiple Nodes from the Graph.

```
delEdges :: [(Int,Int)] → Graph a b → Graph a b
```

Remove multiple Edges from the Graph.

```
isEmpty :: Graph a b → Bool
```

test if the given Graph is empty.

```
match :: Int → Graph a b → (Maybe ([(b,Int)],Int,a,[(b,Int)]),Graph a b)
```

match is the complement side of (:&), decomposing a Graph into the MContext found for the given node and the remaining Graph.

```
noNodes :: Graph a b → Int
```

The number of Nodes in a Graph.

```
nodeRange :: Graph a b → (Int,Int)
```

The minimum and maximum Node in a Graph.

```
context :: Graph a b → Int → ([(b,Int)],Int,a,[(b,Int)])
```

Find the context for the given Node. In contrast to "match", "context" causes an error if the Node is not present in the Graph.

```
lab :: Graph a b → Int → Maybe a
```

Find the label for a Node.

```
neighbors :: Graph a b → Int → [Int]
```

Find the neighbors for a Node.

```
suc :: Graph a b → Int → [Int]
```

Find all Nodes that have a link from the given Node.

```
pre :: Graph a b → Int → [Int]
```

Find all Nodes that link to to the given Node.

```
lsuc :: Graph a b → Int → [(Int,b)]
```

Find all Nodes and their labels, which are linked from the given Node.

`lpre :: Graph a b → Int → [(Int,b)]`

Find all Nodes that link to the given Node and the label of each link.

`out :: Graph a b → Int → [(Int,Int,b)]`

Find all outward-bound LEdges for the given Node.

`inn :: Graph a b → Int → [(Int,Int,b)]`

Find all inward-bound LEdges for the given Node.

`outdeg :: Graph a b → Int → Int`

The outward-bound degree of the Node.

`indeg :: Graph a b → Int → Int`

The inward-bound degree of the Node.

`deg :: Graph a b → Int → Int`

The degree of the Node.

`gelem :: Int → Graph a b → Bool`

True if the Node is present in the Graph.

`equal :: Graph a b → Graph a b → Bool`

graph equality

`node' :: ((a,Int),Int,b,[(a,Int)]) → Int`

The Node in a Context.

`lab' :: ((a,Int),Int,b,[(a,Int)]) → b`

The label in a Context.

`labNode' :: ((a,Int),Int,b,[(a,Int)]) → (Int,b)`

The LNode from a Context.

`neighbors' :: ((a,Int),Int,b,[(a,Int)]) → [Int]`

All Nodes linked to or from in a Context.

`suc' :: ((a,Int),Int,b,[(a,Int)]) → [Int]`

All Nodes linked to in a Context.

`pre' :: ((a,Int),Int,b,[(a,Int)]) → [Int]`

All Nodes linked from in a Context.

$\text{lpre}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow [(\text{Int}, \text{a})]$

All Nodes linked from in a Context, and the label of the links.

$\text{lsuc}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow [(\text{Int}, \text{a})]$

All Nodes linked from in a Context, and the label of the links.

$\text{out}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow [(\text{Int}, \text{Int}, \text{a})]$

All outward-directed LEdges in a Context.

$\text{inn}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow [(\text{Int}, \text{Int}, \text{a})]$

All inward-directed LEdges in a Context.

$\text{outdeg}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow \text{Int}$

The outward degree of a Context.

$\text{indeg}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow \text{Int}$

The inward degree of a Context.

$\text{deg}' :: ([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow \text{Int}$

The degree of a Context.

$\text{labNodes} :: \text{Graph a b} \rightarrow [(\text{Int}, \text{a})]$

A list of all LNodes in the Graph.

$\text{labEdges} :: \text{Graph a b} \rightarrow [(\text{Int}, \text{Int}, \text{b})]$

A list of all LEdges in the Graph.

$\text{nodes} :: \text{Graph a b} \rightarrow [\text{Int}]$

List all Nodes in the Graph.

$\text{edges} :: \text{Graph a b} \rightarrow [(\text{Int}, \text{Int})]$

List all Edges in the Graph.

$\text{newNodes} :: \text{Int} \rightarrow \text{Graph a b} \rightarrow [\text{Int}]$

List N available Nodes, ie Nodes that are not used in the Graph.

$\text{unfold} :: (([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow \text{c} \rightarrow \text{c}) \rightarrow \text{c} \rightarrow \text{Graph b a} \rightarrow \text{c}$

Fold a function over the graph.

$\text{gmap} :: (([(\text{a}, \text{Int})], \text{Int}, \text{b}, [(\text{a}, \text{Int})]) \rightarrow (([\text{c}, \text{Int}], \text{Int}, \text{d}, [(\text{c}, \text{Int})]))) \rightarrow \text{Graph b a} \rightarrow \text{Graph d c}$

Map a function over the graph.

```
nmap :: (a → b) → Graph a c → Graph b c
```

Map a function over the Node labels in a graph.

```
emap :: (a → b) → Graph c a → Graph c b
```

Map a function over the Edge labels in a graph.

```
labUEdges :: [(a,b)] → [(a,b,())]
```

add label () to list of edges (node,node)

```
labUNodes :: [a] → [(a,())]
```

add label () to list of nodes

```
showGraph :: Graph a b → String
```

Represent Graph as String

A.3.5 Library Random

Library for pseudo-random number generation in Curry.

This library provides operations for generating pseudo-random number sequences. For any given seed, the sequences generated by the operations in this module should be **identical** to the sequences generated by the `java.util.Random` package.

The algorithm is taken from http://en.wikipedia.org/wiki/Random_number_generation. There is an assumption that all operations are implicitly executed mod 2^{32} (unsigned 32-bit integers) !!! GHC computes between -2^{29} and $2^{29}-1$, thus the sequence is NOT as random as one would like.

```
m_w = <choose-initializer>; /* must not be zero */  
m_z = <choose-initializer>; /* must not be zero */
```

```
uint get_random()  
{  
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);  
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);  
    return (m_z << 16) + m_w; /* 32-bit result */  
}
```

Exported functions:

```
nextInt :: Int → [Int]
```

Returns a sequence of pseudorandom, integer values.

```
nextIntRange :: Int → Int → [Int]
```

Returns a pseudorandom sequence of values between 0 (inclusive) and the specified value (exclusive).

`nextBoolean :: Int → [Bool]`

Returns a pseudorandom sequence of boolean values.

`getRandomSeed :: IO Int`

Returns a time-dependent integer number as a seed for really random numbers. Should only be used as a seed for pseudorandom number sequence and not as a random number since the precision is limited to milliseconds

A.3.6 Library RedBlackTree

Library with an implementation of red-black trees:

Serves as the base for both TableRBT and SetRBT All the operations on trees are generic, i.e., one has to provide two explicit order predicates ("lessThan" and "eq" below) on elements.

Exported types:

`data RedBlackTree`

A red-black tree consists of a tree structure and three order predicates. These predicates generalize the red black tree. They define 1) equality when inserting into the tree

eg for a set `eqInsert` is `(==)`, for a multiset it is `(-> False)` for a lookUp-table it is `((==) . fst)` 2) equality for looking up values eg for a set `eqLookUp` is `(==)`, for a multiset it is `(==)` for a lookUp-table it is `((==) . fst)` 3) the (less than) relation for the binary search tree

Exported constructors:

Exported functions:

`empty :: (a → a → Bool) → (a → a → Bool) → (a → a → Bool) → RedBlackTree a`

The three relations are inserted into the structure by function `empty`. Returns an empty tree, i.e., an empty red-black tree augmented with the order predicates.

`isEmpty :: RedBlackTree a → Bool`

Test on emptiness

`newTreeLike :: RedBlackTree a → RedBlackTree a`

Creates a new empty red black tree from with the same ordering as a give one.

`lookup :: a → RedBlackTree a → Maybe a`

Returns an element if it is contained in a red-black tree.

`update :: a → RedBlackTree a → RedBlackTree a`

Updates/inserts an element into a RedBlackTree.

```
delete :: a → RedBlackTree a → RedBlackTree a
```

Deletes entry from red black tree.

```
tree2list :: RedBlackTree a → [a]
```

Transforms a red-black tree into an ordered list of its elements.

```
sort :: (a → a → Bool) → [a] → [a]
```

Generic sort based on insertion into red-black trees. The first argument is the order for the elements.

```
setInsertEquivalence :: (a → a → Bool) → RedBlackTree a → RedBlackTree a
```

For compatibility with old version only

A.3.7 Library SCC

Computing strongly connected components

Copyright (c) 2000 - 2003, Wolfgang Lux See LICENSE for the full license.

The function `scc` computes the strongly connected components of a list of entities in two steps. First, the list is topologically sorted "downwards" using the *defines* relation. Then the resulting list is sorted "upwards" using the *uses* relation and partitioned into the connected components. Both relations are computed within this module using the bound and free names of each declaration. In order to avoid useless recomputations, the code in the module first decorates the declarations with their bound and free names and a unique number. The latter is only used to provide a trivial ordering so that the declarations can be used as set elements.

Exported functions:

```
scc :: (a → [b]) → (a → [b]) → [a] → [[a]]
```

Computes the strongly connected components of a list of entities. To be flexible, we distinguish the nodes and the entities defined in this node.

A.3.8 Library SetRBT

Library with an implementation of sets as red-black trees.

All the operations on sets are generic, i.e., one has to provide an explicit order predicate (`<`) (less-than) on elements.

Exported types:

```
type SetRBT a = RedBlackTree a
```

Exported functions:

`emptySetRBT :: (a → a → Bool) → RedBlackTree a`

Returns an empty set, i.e., an empty red-black tree augmented with an order predicate.

`isEmptySetRBT :: RedBlackTree a → Bool`

Test for an empty set.

`elemRBT :: a → RedBlackTree a → Bool`

Returns true if an element is contained in a (red-black tree) set.

`insertRBT :: a → RedBlackTree a → RedBlackTree a`

Inserts an element into a set if it is not already there.

`insertMultiRBT :: a → RedBlackTree a → RedBlackTree a`

Inserts an element into a multiset. Thus, the same element can have several occurrences in the multiset.

`deleteRBT :: a → RedBlackTree a → RedBlackTree a`

delete an element from a set. Deletes only a single element from a multi set

`setRBT2list :: RedBlackTree a → [a]`

Transforms a (red-black tree) set into an ordered list of its elements.

`unionRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a`

Computes the union of two (red-black tree) sets. This is done by inserting all elements of the first set into the second set.

`intersectRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a`

Computes the intersection of two (red-black tree) sets. This is done by inserting all elements of the first set contained in the second set into a new set, which order is taken from the first set.

`sortRBT :: (a → a → Bool) → [a] → [a]`

Generic sort based on insertion into red-black trees. The first argument is the order for the elements.

A.3.9 Library Sort

A collection of useful functions for sorting and comparing characters, strings, and lists.

Exported functions:

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

Quicksort.

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

Bottom-up mergesort.

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

Less-or-equal on lists.

`cmpList :: (a -> a -> Ordering) -> [a] -> [a] -> Ordering`

Comparison of lists.

`leqChar :: Char -> Char -> Bool`

Less-or-equal on characters (deprecated, use `Prelude.<=</code></=</code>`).

`cmpChar :: Char -> Char -> Ordering`

Comparison of characters (deprecated, use `Prelude.compare`).

`leqCharIgnoreCase :: Char -> Char -> Bool`

Less-or-equal on characters ignoring case considerations.

`leqString :: String -> String -> Bool`

Less-or-equal on strings (deprecated, use `Prelude.<=</code></=</code>`).

`cmpString :: String -> String -> Ordering`

Comparison of strings (deprecated, use `Prelude.compare`).

`leqStringIgnoreCase :: String -> String -> Bool`

Less-or-equal on strings ignoring case considerations.

`leqLexGerman :: String -> String -> Bool`

Lexicographical ordering on German strings. Thus, upper/lowercase are not distinguished and Umlauts are sorted as vocals.

A.3.10 Library TableRBT

Library with an implementation of tables as red-black trees:

A table is a finite mapping from keys to values. All the operations on tables are generic, i.e., one has to provide an explicit order predicate ("`cmp`" below) on elements. Each inner node in the red-black tree contains a key-value association.

Exported types:

```
type TableRBT a b = RedBlackTree (a,b)
```

Exported functions:

```
emptyTableRBT :: (a → a → Bool) → RedBlackTree (a,b)
```

Returns an empty table, i.e., an empty red-black tree.

```
isEmptyTable :: RedBlackTree (a,b) → Bool
```

tests whether a given table is empty

```
lookupRBT :: a → RedBlackTree (a,b) → Maybe b
```

Looks up an entry in a table.

```
updateRBT :: a → b → RedBlackTree (a,b) → RedBlackTree (a,b)
```

Inserts or updates an element in a table.

```
tableRBT2list :: RedBlackTree (a,b) → [(a,b)]
```

Transforms the nodes of red-black tree into a list.

```
deleteRBT :: a → RedBlackTree (a,b) → RedBlackTree (a,b)
```

A.3.11 Library Traversal

Library to support lightweight generic traversals through tree-structured data. See here⁷ for a description of the library.

Exported types:

```
type Traversable a b = a → ([b], [b] → a)
```

A datatype is `Traversable` if it defines a function that can decompose a value into a list of children of the same type and recombine new children to a new value of the original type.

⁷<http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html>

Exported functions:

`noChildren :: a → ([b], [b] → a)`

Traversal function for constructors without children.

`children :: (a → ([b], [b] → a)) → a → [b]`

Yields the children of a value.

`replaceChildren :: (a → ([b], [b] → a)) → a → [b] → a`

Replaces the children of a value.

`mapChildren :: (a → ([b], [b] → a)) → (b → b) → a → a`

Applies the given function to each child of a value.

`family :: (a → ([a], [a] → a)) → a → [a]`

Computes a list of the given value, its children, those children, etc.

`childFamilies :: (a → ([b], [b] → a)) → (b → ([b], [b] → b)) → a → [b]`

Computes a list of family members of the children of a value. The value and its children can have different types.

`mapFamily :: (a → ([a], [a] → a)) → (a → a) → a → a`

Applies the given function to each member of the family of a value. Proceeds bottom-up.

`mapChildFamilies :: (a → ([b], [b] → a)) → (b → ([b], [b] → b)) → (b → b) → a → a`

Applies the given function to each member of the families of the children of a value. The value and its children can have different types. Proceeds bottom-up.

`evalFamily :: (a → ([a], [a] → a)) → (a → Maybe a) → a → a`

Applies the given function to each member of the family of a value as long as possible. On each member of the family of the result the given function will yield `Nothing`. Proceeds bottom-up.

`evalChildFamilies :: (a → ([b], [b] → a)) → (b → ([b], [b] → b)) → (b → Maybe b) → a → a`

Applies the given function to each member of the families of the children of a value as long as possible. Similar to `evalFamily`.

`fold :: (a → ([a], [a] → a)) → (a → [b] → b) → a → b`

Implements a traversal similar to a fold with possible default cases.


```
foldChildren :: (a -> ([b],[b] -> a)) -> (b -> ([b],[b] -> b)) -> (a -> [c] -> d)
-> (b -> [c] -> c) -> a -> d
```

Fold the children and combine the results.

```
replaceChildrenIO :: (a -> ([b],[b] -> a)) -> a -> IO [b] -> IO a
```

IO version of replaceChildren

```
mapChildrenIO :: (a -> ([b],[b] -> a)) -> (b -> IO b) -> a -> IO a
```

IO version of mapChildren

```
mapFamilyIO :: (a -> ([a],[a] -> a)) -> (a -> IO a) -> a -> IO a
```

IO version of mapFamily

```
mapChildFamiliesIO :: (a -> ([b],[b] -> a)) -> (b -> ([b],[b] -> b)) -> (b -> IO
b) -> a -> IO a
```

IO version of mapChildFamilies

```
evalFamilyIO :: (a -> ([a],[a] -> a)) -> (a -> IO (Maybe a)) -> a -> IO a
```

IO version of evalFamily

```
evalChildFamiliesIO :: (a -> ([b],[b] -> a)) -> (b -> ([b],[b] -> b)) -> (b -> IO
(Maybe b)) -> a -> IO a
```

IO version of evalChildFamilies

A.3.12 Library Rewriting.Term

Library for representation of first-order terms.

This library is the basis of other libraries for the manipulation of first-order terms, e.g., unification of terms. Therefore, this library also defines other structures, like term equations or positions.

Exported types:

```
type VarIdx = Int
```

Variable index, identifying a variable.

```
type TermEq a = (Term a,Term a)
```

The type of an equation.

```
type TermEqs a = [(Term a,Term a)]
```

The type of multiple equations.

```
type Pos = [Int]
```

A position in a term represented as a list of integers. Arguments are enumerated from 0.

data Term

Representation of a first-order terms. It is parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `TermVar :: Int → Term a`
`TermVar i`
 - The variable with index `i`
- `TermCons :: a → [Term a] → Term a`
`TermCons name args`
 - The constructor with constructor `name` and argument terms `args`

A.3.13 Library `Rewriting.Substitution`

Library for representation of substitutions on first-order terms.

Exported types:

`type Subst a = FM Int (Term a)`

The (abstract) data type for substitutions.

Exported functions:

`showSubst :: FM Int (Term a) → String`

Pretty string representation of a substitution.

`emptySubst :: FM Int (Term a)`

The empty substitution

`extendSubst :: FM Int (Term a) → Int → Term a → FM Int (Term a)`

Extend the substitution with the given mapping.

`lookupSubst :: FM Int (Term a) → Int → Maybe (Term a)`

Searches the substitution for a mapping from the given variable index to a term.

`applySubst :: FM Int (Term a) → Term a → Term a`

Applies a substitution to a single term.

A.3.14 Library Rewriting.Unification

Library for unification on first-order terms.

This library implements a unification algorithm using reference tables.

Exported functions:

```
unify :: [(Term a,Term a)] → Either (UnificationError a) (FM Int (Term a))
```

Unifies the given equations.

A.3.15 Library Rewriting.UnificationSpec

Library for specifying the unification on first-order terms.

This library implements a general unification algorithm. Because the algorithm is easy to understand, but rather slow, it serves as a specification for more elaborate implementations.

Exported types:

```
data UnificationError
```

The data type for the different kinds of errors that can occur during unification.

Exported constructors:

- `Clash :: (Term a) → (Term a) → UnificationError a`

```
Clash t1 t2
```

– Two term constructors with different names are supposed to be equal.

- `OccurCheck :: Int → (Term a) → UnificationError a`

```
OccurCheck v t
```

– A term is supposed to be equal to a term in which it occurs as a subterm.

Exported functions:

```
unify :: [(Term a,Term a)] → Either (UnificationError a) (FM Int (Term a))
```

Unifies the given equations.

A.4 Libraries for Web Applications

A.4.1 Library CategorizedHtmlList

This library provides functions to categorize a list of entities into a HTML page with an index access (e.g., "A-Z") to these entities.

Exported functions:

```
list2CategorizedHtml :: [(a,[HtmlExp])] → [(b,String)] → (a → b → Bool) → [HtmlExp]
```

General categorization of a list of entries.

The item will occur in every category for which the boolean function `categoryFun` yields `True`.

```
categorizeByItemKey :: [(String,[HtmlExp])] → [HtmlExp]
```

Categorize a list of entries with respect to the initial keys.

The categories are named as all initial characters of the keys of the items.

```
stringList2ItemList :: [String] → [(String,[HtmlExp])]
```

Convert a string list into an key-item list. The strings are used as keys and for the simple text layout.

A.4.2 Library HTML

Library for HTML and CGI programming. [This paper](#) contains a description of the basic ideas behind this library.

The installation of a cgi script written with this library can be done by the command

```
makecurrycgi -m initialForm -o /home/joe/public_html/prog.cgi prog
```

where `prog` is the name of the Curry program with the cgi script, `/home/joe/public_html/prog.cgi` is the desired location of the compiled cgi script, and `initialForm` is the Curry expression (of type `IO HtmlForm`) computing the HTML form (where `makecurrycgi` is a shell script stored in `packshome/bin`).

Exported types:

```
type CgiEnv = CgiRef → String
```

The type for representing cgi environments (i.e., mappings from cgi references to the corresponding values of the input elements).

```
type HtmlHandler = (CgiRef → String) → IO HtmlForm
```

The type of event handlers in HTML forms.

```
data CgiRef
```

The (abstract) data type for representing references to input elements in HTML forms.

Exported constructors:

```
data HtmlExp
```

The data type for representing HTML expressions.

Exported constructors:

- `HtmlText :: String → HtmlExp`

`HtmlText s`

– a text string without any further structure

- `HtmlStruct :: String → [(String,String)] → [HtmlExp] → HtmlExp`

`HtmlStruct t as hs`

– a structure with a tag, attributes, and HTML expressions inside the structure

- `HtmlCRef :: HtmlExp → CgiRef → HtmlExp`

`HtmlCRef h ref`

– an input element (described by the first argument) with a cgi reference

- `HtmlEvent :: HtmlExp → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

`HtmlEvent h hdlr`

– an input element (first arg) with an associated event handler (typically, a submit button)

`data HtmlForm`

The data type for representing HTML forms (active web pages) and return values of HTML forms.

Exported constructors:

- `HtmlForm :: String → [FormParam] → [HtmlExp] → HtmlForm`

`HtmlForm t ps hs`

– an HTML form with title `t`, optional parameters (e.g., cookies) `ps`, and contents `hs`

- `HtmlAnswer :: String → String → HtmlForm`

`HtmlAnswer t c`

– an answer in an arbitrary format where `t` is the content type (e.g., "text/plain") and `c` is the contents

`data FormParam`

The possible parameters of an HTML form. The parameters of a cookie (`FormCookie`) are its name and value and optional parameters (expiration date, domain, path (e.g., the path `"/"` makes the cookie valid for all documents on the server), security) which are collected in a list.

Exported constructors:

- `FormCookie :: String → String → [CookieParam] → FormParam`
`FormCookie name value params`
 - a cookie to be sent to the client's browser
- `FormCSS :: String → FormParam`
`FormCSS s`
 - a URL for a CSS file for this form
- `FormJScript :: String → FormParam`
`FormJScript s`
 - a URL for a Javascript file for this form
- `FormOnSubmit :: String → FormParam`
`FormOnSubmit s`
 - a JavaScript statement to be executed when the form is submitted (i.e., `<form ... onsubmit="s">`)
- `FormTarget :: String → FormParam`
`FormTarget s`
 - a name of a target frame where the output of the script should be represented (should only be used for scripts running in a frame)
- `FormEnc :: String → FormParam`
`FormEnc`
 - the encoding scheme of this form
- `FormMeta :: [(String,String)] → FormParam`
`FormMeta as`
 - meta information (in form of attributes) for this form
- `HeadInclude :: HtmlExp → FormParam`
`HeadInclude he`
 - HTML expression to be included in form header
- `MultipleHandlers :: FormParam`
`MultipleHandlers`

- indicates that the event handlers of the form can be multiply used (i.e., are not deleted if the form is submitted so that they are still available when going back in the browser; but then there is a higher risk that the web server process might overflow with unused events); the default is a single use of event handlers, i.e., one cannot use the back button in the browser and submit the same form again (which is usually a reasonable behavior to avoid double submissions of data).

- `BodyAttr :: (String,String) → FormParam`

`BodyAttr ps`

- optional attribute for the body element (more than one occurrence is allowed)

`data CookieParam`

The possible parameters of a cookie.

Exported constructors:

- `CookieExpire :: ClockTime → CookieParam`
- `CookieDomain :: String → CookieParam`
- `CookiePath :: String → CookieParam`
- `CookieSecure :: CookieParam`

`data HtmlPage`

The data type for representing HTML pages. The constructor arguments are the title, the parameters, and the contents (body) of the web page.

Exported constructors:

- `HtmlPage :: String → [PageParam] → [HtmlExp] → HtmlPage`

`data PageParam`

The possible parameters of an HTML page.

Exported constructors:

- `PageEnc :: String → PageParam`

`PageEnc`

- the encoding scheme of this page

- `PageCSS :: String → PageParam`

`PageCSS s`

- a URL for a CSS file for this page

- `PageJScript :: String → PageParam`
`PageJScript s`
 – a URL for a Javascript file for this page
- `PageMeta :: [(String,String)] → PageParam`
`PageMeta as`
 – meta information (in form of attributes) for this page
- `PageLink :: [(String,String)] → PageParam`
`PageLink as`
 – link information (in form of attributes) for this page
- `PageBodyAttr :: (String,String) → PageParam`
`PageBodyAttr attr`
 – optional attribute for the body element of the page (more than one occurrence is allowed)

Exported functions:

`defaultEncoding :: String`

The default encoding used in generated web pages.

`idOfCgiRef :: CgiRef → String`

Internal identifier of a `CgiRef` (intended only for internal use in other libraries!).

`formEnc :: String → FormParam`

An encoding scheme for a HTML form.

`formCSS :: String → FormParam`

A URL for a CSS file for a HTML form.

`formMetaInfo :: [(String,String)] → FormParam`

Meta information for a HTML form. The argument is a list of attributes included in the `meta`-tag in the header for this form.

`formBodyAttr :: (String,String) → FormParam`

Optional attribute for the body element of the HTML form. More than one occurrence is allowed, i.e., all such attributes are collected.

`form :: String → [HtmlExp] → HtmlForm`

A basic HTML form for active web pages with the default encoding and a default background.

`standardForm :: String → [HtmlExp] → HtmlForm`

A standard HTML form for active web pages where the title is included in the body as the first header.

`cookieForm :: String → [(String,String)] → [HtmlExp] → HtmlForm`

An HTML form with simple cookies. The cookies are sent to the client's browser together with this form.

`addCookies :: [(String,String)] → HtmlForm → HtmlForm`

Add simple cookie to HTML form. The cookies are sent to the client's browser together with this form.

`answerText :: String → HtmlForm`

A textual result instead of an HTML form as a result for active web pages.

`answerEncText :: String → String → HtmlForm`

A textual result instead of an HTML form as a result for active web pages where the encoding is given as the first parameter.

`addFormParam :: HtmlForm → FormParam → HtmlForm`

Adds a parameter to an HTML form.

`redirect :: Int → String → HtmlForm → HtmlForm`

Adds redirection to given HTML form.

`expires :: Int → HtmlForm → HtmlForm`

Adds expire time to given HTML form.

`addSound :: String → Bool → HtmlForm → HtmlForm`

Adds sound to given HTML form. The functions adds two different declarations for sound, one invented by Microsoft for the internet explorer, one introduced for netscape. As neither is an official part of HTML, `addsound` might not work on all systems and browsers. The greatest chance is by using sound files in MID-format.

`pageEnc :: String → PageParam`

An encoding scheme for a HTML page.

`pageCSS :: String → PageParam`

A URL for a CSS file for a HTML page.

`pageMetaInfo :: [(String,String)] → PageParam`

Meta information for a HTML page. The argument is a list of attributes included in the `meta`-tag in the header for this page.

`pageLinkInfo :: [(String,String)] → PageParam`

Link information for a HTML page. The argument is a list of attributes included in the `link`-tag in the header for this page.

`pageBodyAttr :: (String,String) → PageParam`

Optional attribute for the body element of the web page. More than one occurrence is allowed, i.e., all such attributes are collected.

`page :: String → [HtmlExp] → HtmlPage`

A basic HTML web page with the default encoding.

`standardPage :: String → [HtmlExp] → HtmlPage`

A standard HTML web page where the title is included in the body as the first header.

`addPageParam :: HtmlPage → PageParam → HtmlPage`

Adds a parameter to an HTML page.

`htxt :: String → HtmlExp`

Basic text as HTML expression. The text may contain special HTML chars (like `<`, `>`, `&`, `”`) which will be quoted so that they appear as in the parameter string.

`htxts :: [String] → [HtmlExp]`

A list of strings represented as a list of HTML expressions. The strings may contain special HTML chars that will be quoted.

`hempty :: HtmlExp`

An empty HTML expression.

`nbspc :: HtmlExp`

Non breaking Space

`h1 :: [HtmlExp] → HtmlExp`

Header 1

`h2 :: [HtmlExp] → HtmlExp`

Header 2

`h3 :: [HtmlExp] → HtmlExp`

Header 3

`h4` :: [HtmlExp] → HtmlExp
Header 4

`h5` :: [HtmlExp] → HtmlExp
Header 5

`par` :: [HtmlExp] → HtmlExp
Paragraph

`section` :: [HtmlExp] → HtmlExp
Section

`header` :: [HtmlExp] → HtmlExp
Header

`footer` :: [HtmlExp] → HtmlExp
Footer

`emphasize` :: [HtmlExp] → HtmlExp
Emphasize

`strong` :: [HtmlExp] → HtmlExp
Strong (more emphasized) text.

`bold` :: [HtmlExp] → HtmlExp
Boldface

`italic` :: [HtmlExp] → HtmlExp
Italic

`nav` :: [HtmlExp] → HtmlExp
Navigation

`code` :: [HtmlExp] → HtmlExp
Program code

`center` :: [HtmlExp] → HtmlExp
Centered text

`blink` :: [HtmlExp] → HtmlExp
Blinking text

`teletype :: [HtmlExp] → HtmlExp`

Teletype font

`pre :: [HtmlExp] → HtmlExp`

Unformatted input, i.e., keep spaces and line breaks and don't quote special characters.

`verbatim :: String → HtmlExp`

Verbatim (unformatted), special characters (<,>,&,") are quoted.

`address :: [HtmlExp] → HtmlExp`

Address

`href :: String → [HtmlExp] → HtmlExp`

Hypertext reference

`anchor :: String → [HtmlExp] → HtmlExp`

An anchored text with a hypertext reference inside a document.

`ulist :: [[HtmlExp]] → HtmlExp`

Unordered list

`olist :: [[HtmlExp]] → HtmlExp`

Ordered list

`litem :: [HtmlExp] → HtmlExp`

A single list item (usually not explicitly used)

`dlist :: [(HtmlExp, HtmlExp)] → HtmlExp`

Description list

`table :: [[[HtmlExp]]] → HtmlExp`

Table with a matrix of items where each item is a list of HTML expressions.

`headedTable :: [[[HtmlExp]]] → HtmlExp`

Similar to `table` but introduces header tags for the first row.

`addHeadings :: HtmlExp → [[HtmlExp]] → HtmlExp`

Add a row of items (where each item is a list of HTML expressions) as headings to a table. If the first argument is not a table, the headings are ignored.

`hrule :: HtmlExp`

Horizontal rule

`breakline :: HtmlExp`

Break a line

`image :: String → String → HtmlExp`

Image

`styleSheet :: String → HtmlExp`

Defines a style sheet to be used in this HTML document.

`style :: String → [HtmlExp] → HtmlExp`

Provides a style for HTML elements. The style argument is the name of a style class defined in a style definition (see `styleSheet`) or in an external style sheet (see form and page parameters `FormCSS` and `PageCSS`).

`textstyle :: String → String → HtmlExp`

Provides a style for a basic text. The style argument is the name of a style class defined in an external style sheet.

`blockstyle :: String → [HtmlExp] → HtmlExp`

Provides a style for a block of HTML elements. The style argument is the name of a style class defined in an external style sheet. This element is used (in contrast to "style") for larger blocks of HTML elements since a line break is placed before and after these elements.

`inline :: [HtmlExp] → HtmlExp`

Joins a list of HTML elements into a single HTML element. Although this construction has no rendering, it is sometimes useful for programming when several HTML elements must be put together.

`block :: [HtmlExp] → HtmlExp`

Joins a list of HTML elements into a block. A line break is placed before and after these elements.

`button :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button with a label string and an event handler

`resetbutton :: String → HtmlExp`

Reset button with a label string

`imageButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button in form of an image.

`textfield :: CgiRef → String → HtmlExp`

Input text field with a reference and an initial contents

```
password :: CgiRef → HtmlExp
```

Input text field (where the entered text is obscured) with a reference

```
textarea :: CgiRef → (Int,Int) → String → HtmlExp
```

Input text area with a reference, height/width, and initial contents

```
checkbox :: CgiRef → String → HtmlExp
```

A checkbox with a reference and a value. The value is returned if checkbox is on, otherwise "" is returned.

```
checkboxchecked :: CgiRef → String → HtmlExp
```

A checkbox that is initially checked with a reference and a value. The value is returned if checkbox is on, otherwise "" is returned.

```
radio_main :: CgiRef → String → HtmlExp
```

A main button of a radio (initially "on") with a reference and a value. The value is returned of this button is on. A complete radio button suite always consists of a main button (*radiomain*) and some further buttons (*radioothers*) with the same reference. Initially, the main button is selected (or nothing is selected if one uses *radiomainoff* instead of *radio_main*). The user can select another button but always at most one button of the radio can be selected. The value corresponding to the selected button is returned in the environment for this radio reference.

```
radio_main_off :: CgiRef → String → HtmlExp
```

A main button of a radio (initially "off") with a reference and a value. The value is returned of this button is on.

```
radio_other :: CgiRef → String → HtmlExp
```

A further button of a radio (initially "off") with a reference (identical to the main button of this radio) and a value. The value is returned of this button is on.

```
selection :: CgiRef → [(String,String)] → HtmlExp
```

A selection button with a reference and a list of name/value pairs. The names are shown in the selection and the value is returned for the selected name.

```
selectionInitial :: CgiRef → [(String,String)] → Int → HtmlExp
```

A selection button with a reference, a list of name/value pairs, and a preselected item in this list. The names are shown in the selection and the value is returned for the selected name.

```
multipleSelection :: CgiRef → [(String,String,Bool)] → HtmlExp
```

A selection button with a reference and a list of name/value/flag pairs. The names are shown in the selection and the value is returned if the corresponding name is selected. If flag is True, the corresponding name is initially selected. If more than one name has been selected, all values are returned in one string where the values are separated by newline (`<n`) characters.

`hiddenfield :: String → String → HtmlExp`

A hidden field to pass a value referenced by a fixed name. This function should be used with care since it may cause conflicts with the CGI-based implementation of this library.

`htmlQuote :: String → String`

Quotes special characters (<, >, &, ", umlauts) in a string as HTML special characters.

`htmlIsoUmlauts :: String → String`

Translates umlauts in iso-8859-1 encoding into HTML special characters.

`addAttr :: HtmlExp → (String, String) → HtmlExp`

Adds an attribute (name/value pair) to an HTML element.

`addAttrs :: HtmlExp → [(String, String)] → HtmlExp`

Adds a list of attributes (name/value pair) to an HTML element.

`addClass :: HtmlExp → String → HtmlExp`

Adds a class attribute to an HTML element.

`showHtmlExps :: [HtmlExp] → String`

Transforms a list of HTML expressions into string representation.

`showHtmlExp :: HtmlExp → String`

Transforms a single HTML expression into string representation.

`showHtmlPage :: HtmlPage → String`

Transforms HTML page into string representation.

`getUrlParameter :: IO String`

Gets the parameter attached to the URL of the script. For instance, if the script is called with URL "http://.../script.cgi?parameter", then "parameter" is returned by this I/O action. Note that an URL parameter should be "URL encoded" to avoid the appearance of characters with a special meaning. Use the functions "urlencoded2string" and "string2urlencoded" to decode and encode such parameters, respectively.

`urlencoded2string :: String → String`

Translates urlencoded string into equivalent ASCII string.

```
string2urlencoded :: String → String
```

Translates arbitrary strings into equivalent urlencoded string.

```
getCookies :: IO [(String,String)]
```

Gets the cookies sent from the browser for the current CGI script. The cookies are represented in the form of name/value pairs since no other components are important here.

```
coordinates :: (CgiRef → String) → Maybe (Int,Int)
```

For image buttons: retrieve the coordinates where the user clicked within the image.

```
runFormServerWithKey :: String → String → IO HtmlForm → IO ()
```

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

```
runFormServerWithKeyAndFormParams :: String → String → [FormParam] → IO  
HtmlForm → IO ()
```

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

```
showLatexExps :: [HtmlExp] → String
```

Transforms HTML expressions into LaTeX string representation.

```
showLatexExp :: HtmlExp → String
```

Transforms an HTML expression into LaTeX string representation.

```
htmlSpecialChars2tex :: String → String
```

Convert special HTML characters into their LaTeX representation, if necessary.

```
showLatexDoc :: [HtmlExp] → String
```

Transforms HTML expressions into a string representation of a complete LaTeX document.

```
showLatexDocWithPackages :: [HtmlExp] → [String] → String
```


Transforms HTML expressions into a string representation of a complete LaTeX document. The variable "packages" holds the packages to add to the latex document e.g. "ngerman"

```
showLatexDocs :: [[HtmlExp]] → String
```

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page.

```
showLatexDocsWithPackages :: [[HtmlExp]] → [String] → String
```

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page. The variable "packages" holds the packages to add to the latex document (e.g., "ngerman").

```
germanLatexDoc :: [HtmlExp] → String
```

show german latex document

```
intForm :: IO HtmlForm → IO ()
```

Execute an HTML form in "interactive" mode.

```
intFormMain :: String → String → String → String → Bool → String → IO  
HtmlForm → IO ()
```

Execute an HTML form in "interactive" mode with various parameters.

A.4.3 Library HtmlCgi

Library to support CGI programming in the HTML library. It is only intended as an auxiliary library to implement dynamic web pages according to the HTML library. It contains a simple script that is installed for a dynamic web page and which sends the user input to the real application server implementing the application.

Exported types:

```
data CgiServerMsg
```

The messages to communicate between the cgi script and the server program. CgiSubmit env cgienv nextpage - pass the environment and show next page, where env are the values of the environment variables of the web script (e.g., *QUERYSTRING*, *REMOTEHOST*, *REMOTE_ADDR*), cgienv are the values in the current form submitted by the client, and nextpage is the answer text to be shown in the next web page

Exported constructors:

- CgiSubmit :: ((String,String)] → ((String,String)] → CgiServerMsg
- GetLoad :: CgiServerMsg
GetLoad

- get info about the current load of the server process
- `SketchStatus :: CgiServerMsg`
`SketchStatus`
 - get a sketch of the status of the server
- `SketchHandlers :: CgiServerMsg`
`SketchHandlers`
 - get a sketch of all event handlers of the server
- `ShowStatus :: CgiServerMsg`
`ShowStatus`
 - show the status of the server with all event handlers
- `CleanServer :: CgiServerMsg`
`CleanServer`
 - clean up the server (with possible termination)
- `StopCgiServer :: CgiServerMsg`
`StopCgiServer`
 - stop the server

Exported functions:

`readCgiServerMsg :: Handle → IO (Maybe CgiServerMsg)`

Reads a line from a handle and check whether it is a syntactically correct cgi server message.

`submitForm :: IO ()`

`runCgiServerCmd :: String → CgiServerMsg → IO ()`

Executes a specific command for a cgi server.

`noHandlerPage :: String → String → String`

`cgiServerRegistry :: String`

The name of the file to register all cgi servers.

`registerCgiServer :: String → String → IO ()`

`unregisterCgiServer :: String → IO ()`

A.4.4 Library HtmlParser

This module contains a very simple parser for HTML documents.

Exported functions:

`readHtmlFile :: String → IO [HtmlExp]`

Reads a file with HTML text and returns the corresponding HTML expressions.

`parseHtmlString :: String → [HtmlExp]`

Transforms an HTML string into a list of HTML expressions. If the HTML string is a well structured document, the list of HTML expressions should contain exactly one element.

A.4.5 Library Mail

This library contains functions for sending emails. The implementation might need to be adapted to the local environment.

Exported types:

`data MailOption`

Options for sending emails.

Exported constructors:

- `CC :: String → MailOption`
CC
– recipient of a carbon copy
- `BCC :: String → MailOption`
BCC
– recipient of a blind carbon copy
- `TO :: String → MailOption`
TO
– recipient of the email

Exported functions:

`sendMail :: String → String → String → String → IO ()`

Sends an email via mailx command.

`sendMailWithOptions :: String → String → [MailOption] → String → IO ()`

Sends an email via mailx command and various options. Note that multiple options are allowed, e.g., more than one CC option for multiple recipient of carbon copies.

Important note: The implementation of this operation is based on the command "mailx" and must be adapted according to your local environment!

A.4.6 Library Markdown

Library to translate [markdown documents](#) into HTML or LaTeX. The slightly restricted subset of the markdown syntax recognized by this implementation is [documented in this page](#).

Exported types:

`type MarkdownDoc = [MarkdownElem]`

A markdown document is a list of markdown elements.

`data MarkdownElem`

The data type for representing the different elements occurring in a markdown document.

Exported constructors:

- `Text :: String → MarkdownElem`

`Text s`

– a simple text in a markdown document

- `Emph :: String → MarkdownElem`

`Emph s`

– an emphasized text in a markdown document

- `Strong :: String → MarkdownElem`

`Strong s`

– a strongly emphasized text in a markdown document

- `Code :: String → MarkdownElem`

`Code s`

– a code string in a markdown document

- `HRef :: String → String → MarkdownElem`
`HRef s u`
 – a reference to URL `u` with text `s` in a markdown document
- `Par :: [MarkdownElem] → MarkdownElem`
`Par md`
 – a paragraph in a markdown document
- `CodeBlock :: String → MarkdownElem`
`CodeBlock s`
 – a code block in a markdown document
- `UList :: [[MarkdownElem]] → MarkdownElem`
`UList mds`
 – an unordered list in a markdown document
- `OList :: [[MarkdownElem]] → MarkdownElem`
`OList mds`
 – an ordered list in a markdown document
- `Quote :: [MarkdownElem] → MarkdownElem`
`Quote md`
 – a quoted paragraph in a markdown document
- `HRule :: MarkdownElem`
`HRule`
 – a hoizontal rule in a markdown document
- `Header :: Int → String → MarkdownElem`
`Header l s`
 – a level `l` header with title `s` in a markdown document

Exported functions:

`fromMarkdownText :: String → [MarkdownElem]`

Parse markdown document from its textual representation.

`removeEscapes :: String → String`

Remove the backlash of escaped markdown characters in a string.

`markdownEscapeChars :: String`

Escape characters supported by markdown.

`markdownText2HTML :: String → [HtmlExp]`

Translate a markdown text into a (partial) HTML document.

`markdownText2CompleteHTML :: String → String → String`

Translate a markdown text into a complete HTML text that can be viewed as a standalone document by a browser. The first argument is the title of the document.

`markdownText2LaTeX :: String → String`

Translate a markdown text into a (partial) LaTeX document. All characters with a special meaning in LaTeX, like dollar or ampersand signs, are quoted.

`markdownText2LaTeXWithFormat :: (String → String) → String → String`

Translate a markdown text into a (partial) LaTeX document where the first argument is a function to translate the basic text occurring in markdown elements to a LaTeX string. For instance, one can use a translation operation that supports passing mathematical formulas in LaTeX style instead of quoting all special characters.

`markdownText2CompleteLaTeX :: String → String`

Translate a markdown text into a complete LaTeX document that can be formatted as a standalone document.

`formatMarkdownInputAsPDF :: IO ()`

Format the standard input (containing markdown text) as PDF.

`formatMarkdownFileAsPDF :: String → IO ()`

Format a file containing markdown text as PDF.

A.4.7 Library URL

Library for dealing with URLs (Uniform Resource Locators).

Exported functions:

`getContentsOfUrl :: String → IO String`

Reads the contents of a document located by a URL. This action requires that the program "wget" is in your path, otherwise the implementation must be adapted to the local installation.

A.4.8 Library WUI

A library to support the type-oriented construction of Web User Interfaces (WUIs).

The ideas behind the application and implementation of WUIs are described in a paper that is available via [this web page](#).

Exported types:

```
type Rendering = [HtmlExp] → HtmlExp
```

A rendering is a function that combines the visualization of components of a data structure into some HTML expression.

```
data WuiHandler
```

A handler for a WUI is an event handler for HTML forms possibly with some specific code attached (for future extensions).

Exported constructors:

```
data WuiSpec
```

The type of WUI specifications. The first component are parameters specifying the behavior of this WUI type (rendering, error message, and constraints on inputs). The second component is a "show" function returning an HTML expression for the edit fields and a WUI state containing the CgiRefs to extract the values from the edit fields. The third component is "read" function to extract the values from the edit fields for a given cgi environment (returned as (Just v)). If the value is not legal, Nothing is returned. The second component of the result contains an HTML edit expression together with a WUI state to edit the value again.

Exported constructors:

```
data WTree
```

A simple tree structure to demonstrate the construction of WUIs for tree types.

Exported constructors:

- WLeaf :: a → WTree a
- WNode :: [WTree a] → WTree a

Exported functions:

```
wuiHandler2button :: String → WuiHandler → HtmlExp
```

Transform a WUI handler into a submit button with a given label string.

```
withRendering :: WuiSpec a → ([HtmlExp] → HtmlExp) → WuiSpec a
```

Puts a new rendering function into a WUI specification.

```
withError :: WuiSpec a → String → WuiSpec a
```

Puts a new error message into a WUI specification.

```
withCondition :: WuiSpec a → (a → Bool) → WuiSpec a
```

Puts a new condition into a WUI specification.

```
transformWSpec :: (a → b, b → a) → WuiSpec a → WuiSpec b
```

Transforms a WUI specification from one type to another.

```
adaptWSpec :: (a → b) → WuiSpec a → WuiSpec b
```

Adapt a WUI specification to a new type. For this purpose, the first argument must be a transformation mapping values from the old type to the new type. This function must be bijective and operationally invertible (i.e., the inverse must be computable by narrowing). Otherwise, use `transformWSpec!`

```
wHidden :: WuiSpec a
```

A hidden widget for a value that is not shown in the WUI. Usually, this is used in components of larger structures, e.g., internal identifiers, data base keys.

```
wConstant :: (a → HtmlExp) → WuiSpec a
```

A widget for values that are shown but cannot be modified. The first argument is a mapping of the value into a HTML expression to show this value.

```
wInt :: WuiSpec Int
```

A widget for editing integer values.

```
wString :: WuiSpec String
```

A widget for editing string values.

```
wStringSize :: Int → WuiSpec String
```

A widget for editing string values with a size attribute.

```
wRequiredString :: WuiSpec String
```

A widget for editing string values that are required to be non-empty.

```
wRequiredStringSize :: Int → WuiSpec String
```

A widget with a size attribute for editing string values that are required to be non-empty.

```
wTextArea :: (Int,Int) → WuiSpec String
```

A widget for editing string values in a text area. The argument specifies the height and width of the text area.

`wSelect :: (a → String) → [a] → WuiSpec a`

A widget to select a value from a given list of values. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into strings to be shown in the selection widget.

`wSelectInt :: [Int] → WuiSpec Int`

A widget to select a value from a given list of integers (provided as the argument). The current value should be contained in the value list and is preselected.

`wSelectBool :: String → String → WuiSpec Bool`

A widget to select a Boolean value via a selection box. The arguments are the strings that are shown for the values True and False in the selection box, respectively.

`wCheckBox :: [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a check box. The first argument are HTML expressions that are shown after the check box. The result is True if the box is checked.

`wMultiCheckSelect :: (a → [HtmlExp]) → [a] → WuiSpec [a]`

A widget to select a list of values from a given list of values via check boxes. The current values should be contained in the value list and are preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the check box.

`wRadioSelect :: (a → [HtmlExp]) → [a] → WuiSpec a`

A widget to select a value from a given list of values via a radio button. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the radio button.

`wRadioBool :: [HtmlExp] → [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a radio button. The arguments are the lists of HTML expressions that are shown after the True and False radio buttons, respectively.

`wPair :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

WUI combinator for pairs.

`wCons2 :: (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c`

WUI combinator for constructors of arity 2. The first argument is the binary constructor. The second and third arguments are the WUI specifications for the argument types.

`wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)`

WUI combinator for triples.

`wCons3 :: (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d`

WUI combinator for constructors of arity 3. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

`w4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)`

WUI combinator for tuples of arity 4.

`wCons4 :: (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e`

WUI combinator for constructors of arity 4. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

`w5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec (a,b,c,d,e)`

WUI combinator for tuples of arity 5.

`wCons5 :: (a → b → c → d → e → f) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f`

WUI combinator for constructors of arity 5. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

`w6Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec (a,b,c,d,e,f)`

WUI combinator for tuples of arity 6.

`wCons6 :: (a → b → c → d → e → f → g) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g`

WUI combinator for constructors of arity 6. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

`w7Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec (a,b,c,d,e,f,g)`

WUI combinator for tuples of arity 7.

`wCons7 :: (a → b → c → d → e → f → g → h) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h`

WUI combinator for constructors of arity 7. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

`w8Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec (a,b,c,d,e,f,g,h)`

WUI combinator for tuples of arity 8.

```
wCons8 :: (a → b → c → d → e → f → g → h → i) → WuiSpec a → WuiSpec b  
→ WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →  
WuiSpec i
```

WUI combinator for constructors of arity 8. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w9Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec (a,b,c,d,e,f,g,h,i)
```

WUI combinator for tuples of arity 9.

```
wCons9 :: (a → b → c → d → e → f → g → h → i → j) → WuiSpec a → WuiSpec  
b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →  
WuiSpec i → WuiSpec j
```

WUI combinator for constructors of arity 9. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w10Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e  
→ WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec  
(a,b,c,d,e,f,g,h,i,j)
```

WUI combinator for tuples of arity 10.

```
wCons10 :: (a → b → c → d → e → f → g → h → i → j → k) → WuiSpec a →  
WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →  
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k
```

WUI combinator for constructors of arity 10. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w11Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →  
WuiSpec (a,b,c,d,e,f,g,h,i,j,k)
```

WUI combinator for tuples of arity 11.

```
wCons11 :: (a → b → c → d → e → f → g → h → i → j → k → l) → WuiSpec a  
→ WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →  
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l
```

WUI combinator for constructors of arity 11. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w12Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →  
WuiSpec l → WuiSpec (a,b,c,d,e,f,g,h,i,j,k,l)
```

WUI combinator for tuples of arity 12.

```
wCons12 :: (a → b → c → d → e → f → g → h → i → j → k → l → m) →  
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →  
WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l →  
WuiSpec m
```

WUI combinator for constructors of arity 12. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wJoinTuple :: WuiSpec a → WuiSpec b → WuiSpec (a,b)
```

WUI combinator to combine two tuples into a joint tuple. It is similar to `wPair` but renders both components as a single tuple provided that the components are already rendered as tuples, i.e., by the rendering function `renderTuple`. This combinator is useful to define combinators for large tuples.

```
wList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are vertically aligned in a table.

```
wListWithHeadings :: [String] → WuiSpec a → WuiSpec [a]
```

Add headings to a standard WUI for list structures:

```
wHList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are horizontally aligned in a table.

```
wMatrix :: WuiSpec a → WuiSpec [[a]]
```

WUI for matrices, i.e., list of list of elements visualized as a matrix.

```
wMaybe :: WuiSpec Bool → WuiSpec a → a → WuiSpec (Maybe a)
```

WUI for Maybe values. It is constructed from a WUI for Booleans and a WUI for the potential values. Nothing corresponds to a selection of False in the Boolean WUI. The value WUI is shown after the Boolean WUI.

```
wCheckMaybe :: WuiSpec a → [HtmlExp] → a → WuiSpec (Maybe a)
```

A WUI for Maybe values where a check box is used to select Just. The value WUI is shown after the check box.

```
wRadioMaybe :: WuiSpec a → [HtmlExp] → [HtmlExp] → a → WuiSpec (Maybe a)
```

A WUI for Maybe values where radio buttons are used to switch between Nothing and Just. The value WUI is shown after the radio button WUI.

```
wEither :: WuiSpec a → WuiSpec b → WuiSpec (Either a b)
```

WUI for union types. Here we provide only the implementation for Either types since other types with more alternatives can be easily reduced to this case.

```
wTree :: WuiSpec a → WuiSpec (WTree a)
```

WUI for tree types. The rendering specifies the rendering of inner nodes. Leaves are shown with their default rendering.

```
renderTuple :: [HtmlExp] → HtmlExp
```

Standard rendering of tuples as a table with a single row. Thus, the elements are horizontally aligned.

```
renderTaggedTuple :: [String] → [HtmlExp] → HtmlExp
```

Standard rendering of tuples with a tag for each element. Thus, each is preceded by a tag, that is set in bold, and all elements are vertically aligned.

```
renderList :: [HtmlExp] → HtmlExp
```

Standard rendering of lists as a table with a row for each item: Thus, the elements are vertically aligned.

```
mainWUI :: WuiSpec a → a → (a → IO HtmlForm) → IO HtmlForm
```

Generates an HTML form from a WUI data specification, an initial value and an update form.

```
wui2html :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp, WuiHandler)
```

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form.

```
wuiInForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO  
HtmlForm) → IO HtmlForm
```

Puts a WUI into a HTML form containing "holes" for the WUI and the handler.

```
wuiWithErrorForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler  
→ IO HtmlForm) → (HtmlExp, WuiHandler)
```

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form. In addition to wui2html, we can provide a skeleton form used to show illegal inputs.

A.4.9 Library WUIjs

A library to support the type-oriented construction of Web User Interfaces (WUIs).

The ideas behind the application and implementation of WUIs are described in a paper that is available via [this web page](#).

In addition to the original library, this version provides also support for JavaScript.

Exported types:

`type Rendering = [HtmlExp] → HtmlExp`

A rendering is a function that combines the visualization of components of a data structure into some HTML expression.

`data WuiHandler`

A handler for a WUI is an event handler for HTML forms possibly with some specific JavaScript code attached.

Exported constructors:

`data WuiSpec`

The type of WUI specifications. The first component are parameters specifying the behavior of this WUI type (rendering, error message, and constraints on inputs). The second component is a "show" function returning an HTML expression for the edit fields and a WUI state containing the CgiRefs to extract the values from the edit fields. The third component is "read" function to extract the values from the edit fields for a given cgi environment (returned as (Just v)). If the value is not legal, Nothing is returned. The second component of the result contains an HTML edit expression together with a WUI state to edit the value again.

Exported constructors:

`data WTree`

A simple tree structure to demonstrate the construction of WUIs for tree types.

Exported constructors:

- `WLeaf :: a → WTree a`
- `WNode :: [WTree a] → WTree a`

Exported functions:

`wuiHandler2button :: String → WuiHandler → HtmlExp`

Transform a WUI handler into a submit button with a given label string.

`withRendering :: WuiSpec a → ([HtmlExp] → HtmlExp) → WuiSpec a`

Puts a new rendering function into a WUI specification.

`withError :: WuiSpec a → String → WuiSpec a`

Puts a new error message into a WUI specification.

`withCondition :: WuiSpec a → (a → Bool) → WuiSpec a`

Puts a new condition into a WUI specification.

```
withConditionJS :: WuiSpec a → (a → Bool) → WuiSpec a
```

Puts a new JavaScript implementation of the condition into a WUI specification.

```
withConditionJSName :: WuiSpec a → (a → Bool,String) → WuiSpec a
```

Puts a new JavaScript implementation of the condition into a WUI specification.

```
transformWSpec :: (a → b,b → a) → WuiSpec a → WuiSpec b
```

Transforms a WUI specification from one type to another.

```
adaptWSpec :: (a → b) → WuiSpec a → WuiSpec b
```

Adapt a WUI specification to a new type. For this purpose, the first argument must be a transformation mapping values from the old type to the new type. This function must be bijective and operationally invertible (i.e., the inverse must be computable by narrowing). Otherwise, use `transformWSpec!`

```
wHidden :: WuiSpec a
```

A hidden widget for a value that is not shown in the WUI. Usually, this is used in components of larger structures, e.g., internal identifiers, data base keys.

```
wConstant :: (a → HtmlExp) → WuiSpec a
```

A widget for values that are shown but cannot be modified. The first argument is a mapping of the value into a HTML expression to show this value.

```
wInt :: WuiSpec Int
```

A widget for editing integer values.

```
wString :: WuiSpec String
```

A widget for editing string values.

```
wStringSize :: Int → WuiSpec String
```

A widget for editing string values with a size attribute.

```
wRequiredString :: WuiSpec String
```

A widget for editing string values that are required to be non-empty.

```
wRequiredStringSize :: Int → WuiSpec String
```

A widget with a size attribute for editing string values that are required to be non-empty.

```
wTextArea :: (Int,Int) → WuiSpec String
```

A widget for editing string values in a text area. The argument specifies the height and width of the text area.

`wSelect :: (a → String) → [a] → WuiSpec a`

A widget to select a value from a given list of values. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into strings to be shown in the selection widget.

`wSelectInt :: [Int] → WuiSpec Int`

A widget to select a value from a given list of integers (provided as the argument). The current value should be contained in the value list and is preselected.

`wSelectBool :: String → String → WuiSpec Bool`

A widget to select a Boolean value via a selection box. The arguments are the strings that are shown for the values True and False in the selection box, respectively.

`wCheckBox :: [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a check box. The first argument are HTML expressions that are shown after the check box. The result is True if the box is checked.

`wMultiCheckSelect :: (a → [HtmlExp]) → [a] → WuiSpec [a]`

A widget to select a list of values from a given list of values via check boxes. The current values should be contained in the value list and are preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the check box.

`wRadioSelect :: (a → [HtmlExp]) → [a] → WuiSpec a`

A widget to select a value from a given list of values via a radio button. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the radio button.

`wRadioBool :: [HtmlExp] → [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a radio button. The arguments are the lists of HTML expressions that are shown after the True and False radio buttons, respectively.

`wJoinTuple :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

WUI combinator to combine two tuples into a joint tuple. It is similar to `wPair` but renders both components as a single tuple provided that the components are already rendered as tuples, i.e., by the rendering function `renderTuple`. This combinator is useful to define combinators for large tuples.

`wPair :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

WUI combinator for pairs.

`wCons2 :: (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c`

WUI combinator for constructors of arity 2. The first argument is the binary constructor. The second and third arguments are the WUI specifications for the argument types.

```
wCons2JS :: Maybe ([JSExp] → JSExp) → (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c
```

```
wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)
```

WUI combinator for triples.

```
wCons3 :: (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d
```

WUI combinator for constructors of arity 3. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons3JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d
```

```
w4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)
```

WUI combinator for tuples of arity 4.

```
wCons4 :: (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e
```

WUI combinator for constructors of arity 4. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons4JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e
```

```
w5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec (a,b,c,d,e)
```

WUI combinator for tuples of arity 5.

```
wCons5 :: (a → b → c → d → e → f) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f
```

WUI combinator for constructors of arity 5. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons5JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f
```

```
w6Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec (a,b,c,d,e,f)
```

WUI combinator for tuples of arity 6.

```
wCons6 :: (a → b → c → d → e → f → g) → WuiSpec a → WuiSpec b → WuiSpec c  
→ WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g
```

WUI combinator for constructors of arity 6. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons6JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g) →  
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →  
WuiSpec g
```

```
w7Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec (a,b,c,d,e,f,g)
```

WUI combinator for tuples of arity 7.

```
wCons7 :: (a → b → c → d → e → f → g → h) → WuiSpec a → WuiSpec b →  
WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h
```

WUI combinator for constructors of arity 7. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons7JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h)  
→ WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →  
WuiSpec g → WuiSpec h
```

```
w8Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec (a,b,c,d,e,f,g,h)
```

WUI combinator for tuples of arity 8.

```
wCons8 :: (a → b → c → d → e → f → g → h → i) → WuiSpec a → WuiSpec b  
→ WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →  
WuiSpec i
```

WUI combinator for constructors of arity 8. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons8JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h →  
i) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f  
→ WuiSpec g → WuiSpec h → WuiSpec i
```

```
w9Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec (a,b,c,d,e,f,g,h,i)
```

WUI combinator for tuples of arity 9.

```
wCons9 :: (a → b → c → d → e → f → g → h → i → j) → WuiSpec a → WuiSpec  
b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →  
WuiSpec i → WuiSpec j
```

WUI combinator for constructors of arity 9. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons9JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h → i  
→ j) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec  
f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j
```

```
w10Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e  
→ WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec  
(a,b,c,d,e,f,g,h,i,j)
```

WUI combinator for tuples of arity 10.

```
wCons10 :: (a → b → c → d → e → f → g → h → i → j → k) → WuiSpec a →  
WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →  
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k
```

WUI combinator for constructors of arity 10. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons10JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h →  
i → j → k) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k
```

```
w11Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →  
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →  
WuiSpec (a,b,c,d,e,f,g,h,i,j,k)
```

WUI combinator for tuples of arity 11.

```
wCons11 :: (a → b → c → d → e → f → g → h → i → j → k → l) → WuiSpec a  
→ WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →  
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l
```

WUI combinator for constructors of arity 11. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons11JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h →
i → j → k → l) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e
→ WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec l
```

```
w12Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec l → WuiSpec (a,b,c,d,e,f,g,h,i,j,k,l)
```

WUI combinator for tuples of arity 12.

```
wCons12 :: (a → b → c → d → e → f → g → h → i → j → k → l → m) →
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →
WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l →
WuiSpec m
```

WUI combinator for constructors of arity 12. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons12JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h
→ i → j → k → l → m) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d →
WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j →
WuiSpec k → WuiSpec l → WuiSpec m
```

```
wList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are vertically aligned in a table.

```
wListWithHeadings :: [String] → WuiSpec a → WuiSpec [a]
```

Add headings to a standard WUI for list structures:

```
wHList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are horizontally aligned in a table.

```
wMatrix :: WuiSpec a → WuiSpec [[a]]
```

WUI for matrices, i.e., list of list of elements visualized as a matrix.

```
wMaybe :: WuiSpec Bool → WuiSpec a → a → WuiSpec (Maybe a)
```

WUI for Maybe values. It is constructed from a WUI for Booleans and a WUI for the potential values. Nothing corresponds to a selection of False in the Boolean WUI. The value WUI is shown after the Boolean WUI.

`wCheckMaybe :: WuiSpec a → [HtmlExp] → a → WuiSpec (Maybe a)`

A WUI for Maybe values where a check box is used to select Just. The value WUI is shown after the check box.

`wRadioMaybe :: WuiSpec a → [HtmlExp] → [HtmlExp] → a → WuiSpec (Maybe a)`

A WUI for Maybe values where radio buttons are used to switch between Nothing and Just. The value WUI is shown after the radio button WUI.

`wEither :: WuiSpec a → WuiSpec b → WuiSpec (Either a b)`

WUI for union types. Here we provide only the implementation for Either types since other types with more alternatives can be easily reduced to this case.

`wTree :: WuiSpec a → WuiSpec (WTree a)`

WUI for tree types. The rendering specifies the rendering of inner nodes. Leaves are shown with their default rendering.

`renderTuple :: [HtmlExp] → HtmlExp`

Standard rendering of tuples as a table with a single row. Thus, the elements are horizontally aligned.

`renderTaggedTuple :: [String] → [HtmlExp] → HtmlExp`

Standard rendering of tuples with a tag for each element. Thus, each is preceded by a tag, that is set in bold, and all elements are vertically aligned.

`renderList :: [HtmlExp] → HtmlExp`

Standard rendering of lists as a table with a row for each item: Thus, the elements are vertically aligned.

`mainWUI :: WuiSpec a → a → (a → IO HtmlForm) → IO HtmlForm`

Generates an HTML form from a WUI data specification, an initial value and an update form.

`wui2html :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp, WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form.

`wuiInForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → IO HtmlForm`

Puts a WUI into a HTML form containing "holes" for the WUI and the handler.

`wuiWithErrorForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → (HtmlExp, WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form. In addition to `wui2html`, we can provide a skeleton form used to show illegal inputs.

A.4.10 Library XML

Library for processing XML data.

Warning: the structure of this library is not stable and might be changed in the future!

Exported types:

data XmlExp

The data type for representing XML expressions.

Exported constructors:

- XText :: String → XmlExp

XText

– a text string (PCDATA)

- XElem :: String → [(String,String)] → [XmlExp] → XmlExp

XElem

– an XML element with tag field, attributes, and a list of XML elements as contents

data Encoding

The data type for encodings used in the XML document.

Exported constructors:

- StandardEnc :: Encoding

- Iso88591Enc :: Encoding

data XmlDocParams

The data type for XML document parameters.

Exported constructors:

- Enc :: Encoding → XmlDocParams

Enc

– the encoding for a document

- DtdUrl :: String → XmlDocParams

DtdUrl

– the url of the DTD for a document

Exported functions:

`tagOf :: XmlExp → String`

Returns the tag of an XML element (or empty for a textual element).

`elemsOf :: XmlExp → [XmlExp]`

Returns the child elements an XML element.

`textOf :: [XmlExp] → String`

Extracts the textual contents of a list of XML expressions. Useful auxiliary function when transforming XML expressions into other data structures.

For instance, `textOf [XText "xy", XElem "a" [] [], XText "bc"] == "xy bc"`

`textOfXml :: [XmlExp] → String`

Included for backward compatibility, better use `textOf!`

`txt :: String → XmlExp`

Basic text (maybe containing special XML chars).

`xml :: String → [XmlExp] → XmlExp`

XML element without attributes.

`writeXmlFile :: String → XmlExp → IO ()`

Writes a file with a given XML document.

`writeXmlFileWithParams :: String → [XmlDocParams] → XmlExp → IO ()`

Writes a file with a given XML document and XML parameters.

`showXmlDoc :: XmlExp → String`

Show an XML document in indented format as a string.

`showXmlDocWithParams :: [XmlDocParams] → XmlExp → String`

`readXmlFile :: String → IO XmlExp`

Reads a file with an XML document and returns the corresponding XML expression.

`readUnsafeXmlFile :: String → IO (Maybe XmlExp)`

Tries to read a file with an XML document and returns the corresponding XML expression, if possible. If file or parse errors occur, `Nothing` is returned.

`readFileWithXmlDocs :: String → IO [XmlExp]`

Reads a file with an arbitrary sequence of XML documents and returns the list of corresponding XML expressions.

```
parseXmlString :: String → [XmlExp]
```

Transforms an XML string into a list of XML expressions. If the XML string is a well structured document, the list of XML expressions should contain exactly one element.

```
updateXmlFile :: (XmlExp → XmlExp) → String → IO ()
```

An action that updates the contents of an XML file by some transformation on the XML document.

A.4.11 Library XmlConv

Provides type-based combinators to construct XML converters. Arbitrary XML data can be represented as algebraic datatypes and vice versa. See [here](#)⁸ for a description of this library.

Exported types:

```
type XmlReads a = ([ (String,String) ], [XmlExp]) → (a, ([ (String,String) ], [XmlExp]))
```

Type of functions that consume some XML data to compute a result

```
type XmlShows a = a → ([ (String,String) ], [XmlExp]) → ([ (String,String) ], [XmlExp])
```

Type of functions that extend XML data corresponding to a given value

```
type XElemConv a = XmlConv Repeatable Elem a
```

Type of converters for XML elements

```
type XAttrConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for attributes

```
type XPrimConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for primitive values

```
type XOptConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for optional values

```
type XRepConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for repetitions

⁸<http://www-ps.informatik.uni-kiel.de/~sebf/projects/xmlconv/>

Exported functions:

`xmlReads :: XmlConv a b c → ((String,String),[XmlExp]) → (c,((String,String),[XmlExp]))`

Takes an XML converter and returns a function that consumes XML data and returns the remaining data along with the result.

`xmlShows :: XmlConv a b c → c → ((String,String),[XmlExp]) → ((String,String),[XmlExp])`

Takes an XML converter and returns a function that extends XML data with the representation of a given value.

`xmlRead :: XmlConv a Elem b → XmlExp → b`

Takes an XML converter and an XML expression and returns a corresponding Curry value.

`xmlShow :: XmlConv a Elem b → b → XmlExp`

Takes an XML converter and a value and returns a corresponding XML expression.

`int :: XmlConv NotRepeatable NoElem Int`

Creates an XML converter for integer values. Integer values must not be used in repetitions and do not represent XML elements.

`float :: XmlConv NotRepeatable NoElem Float`

Creates an XML converter for float values. Float values must not be used in repetitions and do not represent XML elements.

`char :: XmlConv NotRepeatable NoElem Char`

Creates an XML converter for character values. Character values must not be used in repetitions and do not represent XML elements.

`string :: XmlConv NotRepeatable NoElem String`

Creates an XML converter for string values. String values must not be used in repetitions and do not represent XML elements.

`(!) :: XmlConv a b c → XmlConv a b c → XmlConv a b c`

Parallel composition of XML converters.

`element :: String → XmlConv a b c → XmlConv Repeatable Elem c`

Takes an arbitrary XML converter and returns a converter representing an XML element that contains the corresponding data. XML elements may be used in repetitions.

`empty :: a → XmlConv NotRepeatable NoElem a`

Takes a value and returns an XML converter for this value which is not represented as XML data. Empty XML data must not be used in repetitions and does not represent an XML element.

```
attr :: String → (String → a, a → String) → XmlConv NotRepeatable NoElem a
```

Takes a name and string conversion functions and returns an XML converter that represents an attribute. Attributes must not be used in repetitions and do not represent an XML element.

```
adapt :: (a → b, b → a) → XmlConv c d a → XmlConv c d b
```

Converts between arbitrary XML converters for different types.

```
opt :: XmlConv a b c → XmlConv NotRepeatable NoElem (Maybe c)
```

Creates a converter for arbitrary optional XML data. Optional XML data must not be used in repetitions and does not represent an XML element.

```
rep :: XmlConv Repeatable a b → XmlConv NotRepeatable NoElem [b]
```

Takes an XML converter representing repeatable data and returns an XML converter that represents repetitions of this data. Repetitions must not be used in other repetitions and do not represent XML elements.

```
aInt :: String → XmlConv NotRepeatable NoElem Int
```

Creates an XML converter for integer attributes. Integer attributes must not be used in repetitions and do not represent XML elements.

```
aFloat :: String → XmlConv NotRepeatable NoElem Float
```

Creates an XML converter for float attributes. Float attributes must not be used in repetitions and do not represent XML elements.

```
aChar :: String → XmlConv NotRepeatable NoElem Char
```

Creates an XML converter for character attributes. Character attributes must not be used in repetitions and do not represent XML elements.

```
aString :: String → XmlConv NotRepeatable NoElem String
```

Creates an XML converter for string attributes. String attributes must not be used in repetitions and do not represent XML elements.

```
aBool :: String → String → String → XmlConv NotRepeatable NoElem Bool
```

Creates an XML converter for boolean attributes. Boolean attributes must not be used in repetitions and do not represent XML elements.

```
eInt :: String → XmlConv Repeatable Elem Int
```

Creates an XML converter for integer elements. Integer elements may be used in repetitions.

`eFloat :: String → XmlConv Repeatable Elem Float`

Creates an XML converter for float elements. Float elements may be used in repetitions.

`eChar :: String → XmlConv Repeatable Elem Char`

Creates an XML converter for character elements. Character elements may be used in repetitions.

`eString :: String → XmlConv Repeatable Elem String`

Creates an XML converter for string elements. String elements may be used in repetitions.

`eBool :: String → String → XmlConv Repeatable Elem Bool`

Creates an XML converter for boolean elements. Boolean elements may be used in repetitions.

`eEmpty :: String → a → XmlConv Repeatable Elem a`

Takes a name and a value and creates an empty XML element that represents the given value. The created element may be used in repetitions.

`eOpt :: String → XmlConv a b c → XmlConv Repeatable Elem (Maybe c)`

Creates an XML converter that represents an element containing optional XML data. The created element may be used in repetitions.

`eRep :: String → XmlConv Repeatable a b → XmlConv Repeatable Elem [b]`

Creates an XML converter that represents an element containing repeated XML data. The created element may be used in repetitions.

`seq1 :: (a → b) → XmlConv c d a → XmlConv c NoElem b`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq1 :: (a → b) → XmlConv Repeatable c a → XmlConv NotRepeatable NoElem [b]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions but does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq1 :: String → (a → b) → XmlConv c d a → XmlConv Repeatable Elem b`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq1 :: String → (a → b) → XmlConv Repeatable c a → XmlConv Repeatable Elem [b]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq2 :: (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv NotRepeatable NoElem c
```

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

```
repSeq2 :: (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv NotRepeatable NoElem [c]
```

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

```
eSeq2 :: String → (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv Repeatable Elem c
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq2 :: String → (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv Repeatable Elem [c]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq3 :: (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv NotRepeatable NoElem d
```

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

```
repSeq3 :: (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv NotRepeatable NoElem [d]
```

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

```
eSeq3 :: String → (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv Repeatable Elem d
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq3 :: String → (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv Repeatable Elem [d]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq4 :: (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv NotRepeatable NoElem e
```

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

```
repSeq4 :: (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv NotRepeatable NoElem [e]
```

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

```
eSeq4 :: String → (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv Repeatable Elem e
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq4 :: String → (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv Repeatable Elem [e]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq5 :: (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b → XmlConv k l c → XmlConv m n d → XmlConv o p e → XmlConv NotRepeatable NoElem f
```

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

```
repSeq5 :: (a → b → c → d → e → f) → XmlConv Repeatable g a → XmlConv Repeatable h b → XmlConv Repeatable i c → XmlConv Repeatable j d → XmlConv Repeatable k e → XmlConv NotRepeatable NoElem [f]
```

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

```
eSeq5 :: String → (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b  
→ XmlConv k l c → XmlConv m n d → XmlConv o p e → XmlConv Repeatable Elem f
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq5 :: String → (a → b → c → d → e → f) → XmlConv Repeatable g a →  
XmlConv Repeatable h b → XmlConv Repeatable i c → XmlConv Repeatable j d →  
XmlConv Repeatable k e → XmlConv Repeatable Elem [f]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq6 :: (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j k b →  
XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv  
NotRepeatable NoElem g
```

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

```
repSeq6 :: (a → b → c → d → e → f → g) → XmlConv Repeatable h a → XmlConv  
Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d → XmlConv  
Repeatable l e → XmlConv Repeatable m f → XmlConv NotRepeatable NoElem [g]
```

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

```
eSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j  
k b → XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv  
Repeatable Elem g
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

```
eRepSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv Repeatable h a  
→ XmlConv Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d →  
XmlConv Repeatable l e → XmlConv Repeatable m f → XmlConv Repeatable Elem [g]
```

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

A.5 Libraries for Meta-Programming

A.5.1 Library `AbstractCurry.Types`

This library contains a definition for representing Curry programs in Curry and an I/O action to read Curry programs and transform them into this abstract representation.

Note this defines a slightly new format for `AbstractCurry` in comparison to the first proposal of 2003.

Assumption: an abstract Curry program is stored in file with extension `.acy`

Exported types:

```
type MName = String
```

A module name.

```
type QName = (String,String)
```

The data type for representing qualified names. In `AbstractCurry` all names are qualified to avoid name clashes. The first component is the module name and the second component the unqualified name as it occurs in the source program. An exception are locally defined names where the module name is the empty string (to avoid name clashes with a globally defined name).

```
type CVarIName = (Int,String)
```

The type for representing type variables. They are represented by (i,n) where i is a type variable index which is unique inside a function and n is a name (if possible, the name written in the source program).

```
type CField a = ((String,String),a)
```

Labeled record fields

```
type Arity = Int
```

Function arity

```
type CVarIName = (Int,String)
```

Data types for representing object variables. Object variables occurring in expressions are represented by $(\text{Var } i)$ where i is a variable index.

```
data CVisibility
```

Data type to specify the visibility of various entities.

Exported constructors:

- `Public :: CVisibility`
- `Private :: CVisibility`

data CurryProg

Data type for representing a Curry module in the intermediate form. A value of this data type has the form

(CProg modname imports typedecls functions opdecls)

where modname: name of this module, imports: list of modules names that are imported, typedecls: Type declarations functions: Function declarations opdecls: Operator precedence declarations

Exported constructors:

- CurryProg :: String → [String] → [CTypeDecl] → [CFuncDecl] → [COpDecl] → CurryProg

data CTypeDecl

Data type for representing definitions of algebraic data types and type synonyms.
A data type definition of the form

data t x1...xn = ... | c t1...tkc |...

is represented by the Curry term

(CType t v [i1,...,in] [...(CCons c kc v [t1,...,tkc])...])

where each i_j is the index of the type variable x_j .

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- CType :: (String,String) → CVisibility → [(Int,String)] → [CConsDecl] → CTypeDecl
- CTypeSyn :: (String,String) → CVisibility → [(Int,String)] → CTypeExpr → CTypeDecl
- CNewType :: (String,String) → CVisibility → [(Int,String)] → CConsDecl → CTypeDecl

data CConsDecl

A constructor declaration consists of the name of the constructor and a list of the argument types of the constructor. The arity equals the number of types.

Exported constructors:

- `CCons :: (String,String) → CVisibility → [CTypeExpr] → CConsDecl`
- `CRecord :: (String,String) → CVisibility → [CFieldDecl] → CConsDecl`

`data CFieldDecl`

A record field declaration consists of the name of the the label, the visibility and its corresponding type.

Exported constructors:

- `CField :: (String,String) → CVisibility → CTypeExpr → CFieldDecl`

`data CTypeExpr`

Type expression. A type expression is either a type variable, a function type, or a type constructor application.

Note: the names of the predefined type constructors are "Int", "Float", "Bool", "Char", "IO", "()" (unit type), "(...)" (tuple types), "[]" (list type)

Exported constructors:

- `CTVar :: (Int,String) → CTypeExpr`
- `CFuncType :: CTypeExpr → CTypeExpr → CTypeExpr`
- `CTCons :: (String,String) → [CTypeExpr] → CTypeExpr`

`data COpDecl`

Data type for operator declarations. An operator declaration "fix p n" in Curry corresponds to the AbstractCurry term (COp n fix p).

Exported constructors:

- `COp :: (String,String) → CFixity → Int → COpDecl`

`data CFixity`

Data type for operator associativity

Exported constructors:

- `CInfixOp :: CFixity`
- `CInfixlOp :: CFixity`

- `CInfixOp :: CFixity`

`data CFuncDecl`

Data type for representing function declarations.

A function declaration in `AbstractCurry` is a term of the form

`(CFunc name arity visibility type (CRules eval [CRule rule1,...,rulek]))`

and represents the function `name` defined by the rules `rule1,...,rulek`.

Note: the variable indices are unique inside each rule

Thus, a function declaration consists of the name, arity, type, and a list of rules.

A function declaration with the constructor `CmtFunc` is similarly to `CFunc` but has a comment as an additional first argument. This comment could be used by pretty printers that generate a readable Curry program containing documentation comments.

Exported constructors:

- `CFunc :: (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl`
- `CmtFunc :: String → (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl`

`data CRule`

The general form of a function rule. It consists of a list of patterns (left-hand side) and the right-hand side for these patterns.

Exported constructors:

- `CRule :: [CPattern] → CRhs → CRule`

`data CRhs`

Right-hand-side of a `CRule` or a `case` expression. It is either a simple unconditional right-hand side or a list of guards with their corresponding right-hand sides, and a list of local declarations.

Exported constructors:

- `CSimpleRhs :: CExpr → [CLocalDecl] → CRhs`
- `CGuardedRhs :: [(CExpr,CExpr)] → [CLocalDecl] → CRhs`

`data CLocalDecl`

Data type for representing local (let/where) declarations

Exported constructors:

- CLocalFunc :: CFuncDecl → CLocalDecl
- CLocalPat :: CPattern → CRhs → CLocalDecl
- CLocalVars :: [(Int,String)] → CLocalDecl

data CPattern

Data type for representing pattern expressions.

Exported constructors:

- CVar :: (Int,String) → CPattern
- CPLit :: CLiteral → CPattern
- CPComb :: (String,String) → [CPattern] → CPattern
- CPAs :: (Int,String) → CPattern → CPattern
- CPFuncComb :: (String,String) → [CPattern] → CPattern
- CPLazy :: CPattern → CPattern
- CPreRecord :: (String,String) → [((String,String),CPattern)] → CPattern

data CExpr

Data type for representing Curry expressions.

Exported constructors:

- CVar :: (Int,String) → CExpr
- CLit :: CLiteral → CExpr
- CSymbol :: (String,String) → CExpr
- CApply :: CExpr → CExpr → CExpr
- CLambda :: [CPattern] → CExpr → CExpr
- CLetDecl :: [CLocalDecl] → CExpr → CExpr
- CDoExpr :: [CStatement] → CExpr
- CListComp :: CExpr → [CStatement] → CExpr
- CCase :: CCaseType → CExpr → [(CPattern,CRhs)] → CExpr
- CTyped :: CExpr → CTypeExpr → CExpr
- CRecConstr :: (String,String) → [((String,String),CExpr)] → CExpr

- `CRecUpdate :: CExpr → [(String,String),CExpr] → CExpr`

`data CLiteral`

Data type for representing literals occurring in an expression. It is either an integer, a float, or a character constant.

Exported constructors:

- `CIntc :: Int → CLiteral`
- `CFloatc :: Float → CLiteral`
- `CCharc :: Char → CLiteral`
- `CStringc :: String → CLiteral`

`data CStatement`

Data type for representing statements in do expressions and list comprehensions.

Exported constructors:

- `CSEExpr :: CExpr → CStatement`
- `CSPat :: CPattern → CExpr → CStatement`
- `CSLet :: [CLocalDecl] → CStatement`

`data CCaseType`

Type of case expressions

Exported constructors:

- `CRigid :: CCaseType`
- `CFlex :: CCaseType`

Exported functions:

`version :: String`

Current version of AbstractCurry

`pre :: String → (String,String)`

Converts a string into a qualified name of the Prelude.

A.5.2 Library AbstractCurry.Files

This library defines various I/O actions to read Curry programs and transform them into the AbstractCurry representation and to write AbstractCurry files.

Assumption: an abstract Curry program is stored in file with extension `.acy` in the subdirectory `.curry`

Exported functions:

`readCurry :: String → IO CurryProg`

I/O action which parses a Curry program and returns the corresponding typed Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readCurryWithImports :: String → IO [CurryProg]`

Read an AbstractCurry file with all its imports.

`tryReadCurryWithImports :: String → IO (Either [String] [CurryProg])`

`tryReadCurryFile :: String → IO (Either String CurryProg)`

`tryParse :: String → IO (Either String CurryProg)`

Try to parse an AbstractCurry file.

`readUntypedCurry :: String → IO CurryProg`

I/O action which parses a Curry program and returns the corresponding untyped Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads a typed Curry program from a file (with extension ".acy") with respect to some parser options. This I/O action is used by the standard action `readCurry`. It is currently predefined only in `Curry2Prolog`.

`readUntypedCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads an untyped Curry program from a file (with extension ".uacy") with respect to some parser options. For more details see function `readCurryWithParseOptions`

`abstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding AbstractCurry program.

`untypedAbstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding untyped AbstractCurry program.

`readAbstractCurryFile :: String → IO CurryProg`

I/O action which reads an `AbstractCurry` program from a file in `".acy"` format. In contrast to `readCurry`, this action does not parse a source program. Thus, the argument must be the name of an existing file (with suffix `".acy"`) containing an `AbstractCurry` program in `".acy"` format and the result is a Curry term representing this program. It is currently predefined only in `Curry2Prolog`.

`tryReadACYFile :: String → IO (Maybe CurryProg)`

Tries to read an `AbstractCurry` file and returns

- Left `err` , where `err` specifies the error occurred
- Right `prog`, where `prog` is the `AbstractCurry` program

`writeAbstractCurryFile :: String → CurryProg → IO ()`

Writes an `AbstractCurry` program into a file in `".acy"` format. The first argument must be the name of the target file (with suffix `".acy"`).

A.5.3 Library `AbstractCurry.Select`

This library provides some useful operations to select components in `AbstractCurry` programs, i.e., it provides a collection of selector functions for `AbstractCurry`.

Exported functions:

`imports :: CurryProg → [String]`

Returns the imports (module names) of a given curry program.

`functions :: CurryProg → [CFuncDecl]`

Returns the function declarations of a given curry program.

`constructors :: CurryProg → [CConsDecl]`

Returns all constructors of given curry program.

`types :: CurryProg → [CTypeDecl]`

Returns the type declarations of a given curry program.

`publicFuncNames :: CurryProg → [(String,String)]`

Returns the names of all visible functions in given curry program.

`publicConsNames :: CurryProg → [(String,String)]`

Returns the names of all visible constructors in given curry program.

`publicTypeNames :: CurryProg → [(String,String)]`

Returns the names of all visible types in given curry program.

`typeName :: CTypeDecl → (String,String)`

Returns the name of a given type declaration

`typeVis :: CTypeDecl → CVisibility`

Returns the visibility of a given type declaration

`typeCons :: CTypeDecl → [CConsDecl]`

Returns the constructors of a given type declaration.

`consName :: CConsDecl → (String,String)`

Returns the name of a given constructor declaration.

`consVis :: CConsDecl → CVisibility`

Returns the visibility of a given constructor declaration.

`isBaseType :: CTypeExpr → Bool`

Returns true if the type expression is a base type.

`isPolyType :: CTypeExpr → Bool`

Returns true if the type expression contains type variables.

`isFunctionType :: CTypeExpr → Bool`

Returns true if the type expression is a functional type.

`isIOType :: CTypeExpr → Bool`

Returns true if the type expression is (IO t).

`isIOReturnType :: CTypeExpr → Bool`

Returns true if the type expression is (IO t) with t/=() and t is not functional

`argTypes :: CTypeExpr → [CTypeExpr]`

Returns all argument types from a functional type

`resultType :: CTypeExpr → CTypeExpr`

Return the result type from a (nested) functional type

`tvarsOfType :: CTypeExpr → [(Int,String)]`

Returns all type variables occurring in a type expression.

`modsOfType :: CTypeExpr → [String]`

Returns all modules used in the given type.

`funcName :: CFuncDecl → (String,String)`
 Returns the name of a given function declaration

`funcVis :: CFuncDecl → CVisibility`
 Returns the visibility of a given function declaration

`varsOfPat :: CPattern → [(Int,String)]`
 Returns list of all variables occurring in a pattern. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfExp :: CExpr → [(Int,String)]`
 Returns list of all variables occurring in an expression. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfRhs :: CRhs → [(Int,String)]`
 Returns list of all variables occurring in a right-hand side. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfStat :: CStatement → [(Int,String)]`
 Returns list of all variables occurring in a statement. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfLDecl :: CLocalDecl → [(Int,String)]`
 Returns list of all variables occurring in a local declaration. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfFDecl :: CFuncDecl → [(Int,String)]`
 Returns list of all variables occurring in a function declaration. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfRule :: CRule → [(Int,String)]`
 Returns list of all variables occurring in a rule. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`funcNamesOfLDecl :: CLocalDecl → [(String,String)]`

`funcNamesOfFDecl :: CFuncDecl → [(String,String)]`

`funcNamesOfStat :: CStatement → [(String,String)]`

`ldeclsOfRule :: CRule → [CLocalDecl]`

`isPrelude :: String → Bool`
 Tests whether a module name is the prelude.

A.5.4 Library `AbstractCurry.Build`

This library provides some useful operations to write programs that generate `AbstractCurry` programs in a more compact and readable way.

Exported functions:

`(~>)` :: `CTypeExpr` → `CTypeExpr` → `CTypeExpr`

A function type.

`baseType` :: `(String,String)` → `CTypeExpr`

A base type.

`listType` :: `CTypeExpr` → `CTypeExpr`

Constructs a list type from an element type.

`tupleType` :: `[CTypeExpr]` → `CTypeExpr`

Constructs a tuple type from list of component types.

`ioType` :: `CTypeExpr` → `CTypeExpr`

Constructs an IO type from a type.

`maybeType` :: `CTypeExpr` → `CTypeExpr`

Constructs a Maybe type from element type.

`stringType` :: `CTypeExpr`

The type expression of the `String` type.

`intType` :: `CTypeExpr`

The type expression of the `Int` type.

`floatType` :: `CTypeExpr`

The type expression of the `Float` type.

`boolType` :: `CTypeExpr`

The type expression of the `Bool` type.

`unitType` :: `CTypeExpr`

The type expression of the unit type.

`dateType` :: `CTypeExpr`

The type expression of the `Time.CalendarTime` type.

`cfunc :: (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl`

Constructs a function declaration from a given qualified function name, arity, visibility, type expression and list of defining rules.

`cmtfunc :: String → (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl`

Constructs a function declaration from a given comment, qualified function name, arity, visibility, type expression and list of defining rules.

`simpleRule :: [CPattern] → CExpr → CRule`

Constructs a simple rule with a pattern list and an unconditional right-hand side.

`guardedRule :: [CPattern] → [(CExpr,CExpr)] → [CLocalDecl] → CRule`

Constructs a rule with a possibly guarded right-hand side and local declarations. A simple right-hand side is constructed if there is only one True condition.

`noGuard :: CExpr → (CExpr,CExpr)`

Constructs a guarded expression with the trivial guard.

`applyF :: (String,String) → [CExpr] → CExpr`

An application of a qualified function name to a list of arguments.

`applyE :: CExpr → [CExpr] → CExpr`

An application of an expression to a list of arguments.

`constF :: (String,String) → CExpr`

A constant, i.e., an application without arguments.

`applyV :: (Int,String) → [CExpr] → CExpr`

An application of a variable to a list of arguments.

`applyJust :: CExpr → CExpr`

`applyMaybe :: CExpr → CExpr → CExpr → CExpr`

`tupleExpr :: [CExpr] → CExpr`

Constructs a tuple expression from list of component expressions.

`cBranch :: CPattern → CExpr → (CPattern,CRhs)`

Constructs from a pattern and an expression a branch for a case expression.

`tuplePattern :: [CPattern] → CPattern`

Constructs a tuple pattern from list of component patterns.

`pVars :: Int → [CPattern]`

Constructs, for given n, a list of n PVars starting from 0.

`pChar :: Char → CPattern`

Converts a character into a pattern.

`pNil :: CPattern`

Constructs an empty list pattern.

`listPattern :: [CPattern] → CPattern`

Constructs a list pattern from list of component patterns.

`stringPattern :: String → CPattern`

Converts a string into a pattern representing this string.

`list2ac :: [CExpr] → CExpr`

Converts a list of AbstractCurry expressions into an AbstractCurry representation of this list.

`cChar :: Char → CExpr`

Converts a character into an AbstractCurry expression.

`string2ac :: String → CExpr`

Converts a string into an AbstractCurry representation of this string.

`toVar :: Int → CExpr`

Converts an index i into a variable named xi.

`cvar :: String → CExpr`

Converts a string into a variable with index 1.

`cpvar :: String → CPattern`

Converts a string into a pattern variable with index 1.

`ctvar :: String → CTypeExpr`

Converts a string into a type variable with index 1.

A.5.5 Library `AbstractCurry.Pretty`

Pretty-printing of `AbstractCurry`.

This library provides a pretty-printer for `AbstractCurry` modules.

Exported types:

`data Qualification`

Exported constructors:

- `Full :: Qualification`
- `Imports :: Qualification`
- `OnDemand :: Qualification`
- `None :: Qualification`

`data LayoutChoice`

Exported constructors:

- `PreferNestedLayout :: LayoutChoice`
- `PreferFilledLayout :: LayoutChoice`

`data Options`

Exported constructors:

Exported functions:

`defaultOptions :: Options`

`defaultOptions = options 78 2 Imports "" []`. Therefore use these options only with functions like `prettyCurryProg` or `ppCurryProg`, because they will overwrite the module name anyway. Also avoid setting on demand qualification, because no information about related modules is given.

`options :: Int → Int → Qualification → String → [CurryProg] → Options`

`setPageWidth :: Int → Options → Options`

Sets the page width of the pretty printer options.

`setIndentWith :: Int → Options → Options`

Sets the indentation width of the pretty printer options.

`setQualification :: Qualification → Options → Options`

Sets the qualification method to be used by the pretty printer.

`setModName :: String → Options → Options`

Sets the name of the current module in the pretty printer options.

`setLayoutChoice :: LayoutChoice → Options → Options`

Sets the preferred layout in the pretty printer options.

`showCProg :: CurryProg → String`

Shows a pretty formatted version of an abstract Curry Program. The options for pretty-printing are the `defaultOptions`.

`prettyCurryProg :: Options → CurryProg → String`

pretty-print the document generated by `ppCurryProg`, using the page width specified by given options.

`ppCurryProg :: Options → CurryProg → Doc`

pretty-print a `CurryProg` (the representation of a program, written in curry, using `AbstractCurry`) according to given options. This function will overwrite the module name given by options with the name encapsulated in `CurryProg`.

`ppMName :: String → Doc`

pretty-print a module name (just a string).

`ppExports :: Options → [CTypeDecl] → [CFuncDecl] → Doc`

pretty-print exports, i.e. all type and function declarations which are public. extract the type and function declarations which are public and gather their qualified names in a list.

`ppImports :: Options → [String] → Doc`

pretty-print imports (list of module names) by prepending the word "import" to the module name.

`ppCOpDecl :: Options → COpDecl → Doc`

pretty-print operator precedence declarations.

`ppCTypeDecl :: Options → CTypeDecl → Doc`

pretty-print type declarations, like `data ... = ...`, `type ... = ...` or `newtype ... = ...`.

`ppCFuncDecl :: Options → CFuncDecl → Doc`

pretty-print a function declaration.

`ppCFuncDeclWithoutSig :: Options → CFuncDecl → Doc`

pretty-print a function declaration without signature.

`ppCFuncSignature :: Options → (String,String) → CTypeExpr → Doc`

pretty-print a function signature according to given options.

`ppCTypeExpr :: Options → CTypeExpr → Doc`

pretty-print a type expression.

`ppCPattern :: Options → CPattern → Doc`

pretty-print a pattern expression.

`ppCLiteral :: Options → CLiteral → Doc`

pretty-print given literal (Int, Float, ...).

`ppCExpr :: Options → CExpr → Doc`

pretty-print an expression.

`ppCStatement :: Options → CStatement → Doc`

`ppQFunc :: Options → (String,String) → Doc`

pretty-print a function name or constructor name qualified according to given options.
Use `ppQType` or `ppType` for pretty-printing type names.

`ppFunc :: (String,String) → Doc`

pretty-print a function name or constructor name non-qualified. Use `ppQType` or `ppType` for pretty-printing type names.

`ppQType :: Options → (String,String) → Doc`

pretty-print a type (QName) qualified according to given options.

`ppType :: (String,String) → Doc`

pretty-print a type (QName) non-qualified.

A.5.6 Library FlatCurry.Types

This library supports meta-programming, i.e., the manipulation of Curry programs in Curry. For this purpose, the library contains definitions of data types for the representation of so-called FlatCurry programs.

Exported types:

```
type QName = (String,String)
```

The data type for representing qualified names. In FlatCurry all names are qualified to avoid name clashes. The first component is the module name and the second component the unqualified name as it occurs in the source program.

```
type TVarIndex = Int
```

The data type for representing type variables. They are represented by `(TVar i)` where `i` is a type variable index.

```
type VarIndex = Int
```

Data type for representing object variables. Object variables occurring in expressions are represented by `(Var i)` where `i` is a variable index.

```
data Prog
```

Data type for representing a Curry module in the intermediate form. A value of this data type has the form

```
(Prog modname imports typedecls functions opdecls)
```

where `modname` is the name of this module, `imports` is the list of modules names that are imported, and `typedecls`, `functions`, and `opdecls` are the list of data type, function, and operator declarations contained in this module, respectively.

Exported constructors:

- `Prog :: String → [String] → [TypeDecl] → [FuncDecl] → [OpDecl] → Prog`

```
data Visibility
```

Data type to specify the visibility of various entities.

Exported constructors:

- `Public :: Visibility`
- `Private :: Visibility`

```
data TypeDecl
```

Data type for representing definitions of algebraic data types and type synonyms.

A data type definition of the form

```
data t x1...xn = ... | c t1...tkc |...
```

is represented by the FlatCurry term

```
(Type t [i1,...,in] [...(Cons c kc [t1,...,tkc])...])
```

where each i_j is the index of the type variable x_j .

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- `Type :: (String,String) → Visibility → [Int] → [ConsDecl] → TypeDecl`
- `TypeSyn :: (String,String) → Visibility → [Int] → TypeExpr → TypeDecl`

```
data ConsDecl
```

A constructor declaration consists of the name and arity of the constructor and a list of the argument types of the constructor.

Exported constructors:

- `Cons :: (String,String) → Int → Visibility → [TypeExpr] → ConsDecl`

```
data TypeExpr
```

Data type for type expressions. A type expression is either a type variable, a function type, or a type constructor application.

Note: the names of the predefined type constructors are "Int", "Float", "Bool", "Char", "IO", "()" (unit type), "(,...)" (tuple types), "[]" (list type)

Exported constructors:

- `TVar :: Int → TypeExpr`
- `FuncType :: TypeExpr → TypeExpr → TypeExpr`
- `TCons :: (String,String) → [TypeExpr] → TypeExpr`

```
data OpDecl
```


Data type for operator declarations. An operator declaration `fix p n` in Curry corresponds to the FlatCurry term `(Op n fix p)`.

Exported constructors:

- `Op :: (String,String) → Fixity → Int → OpDecl`

`data Fixity`

Data types for the different choices for the fixity of an operator.

Exported constructors:

- `InfixOp :: Fixity`
- `InfixlOp :: Fixity`
- `InfixrOp :: Fixity`

`data FuncDecl`

Data type for representing function declarations.

A function declaration in FlatCurry is a term of the form

`(Func name k type (Rule [i1,...,ik] e))`

and represents the function `name` with definition

`name :: type`
`name x1...xk = e`

where each `ij` is the index of the variable `xj`.

Note: the variable indices are unique inside each function declaration and are usually numbered from 0

External functions are represented as

`(Func name arity type (External s))`

where `s` is the external name associated to this function.

Thus, a function declaration consists of the name, arity, type, and rule.

Exported constructors:

- `Func :: (String,String) → Int → Visibility → TypeExpr → Rule → FuncDecl`

`data Rule`

A rule is either a list of formal parameters together with an expression or an "External" tag.

Exported constructors:

- `Rule :: [Int] → Expr → Rule`
- `External :: String → Rule`

`data CaseType`

Data type for classifying case expressions. Case expressions can be either flexible or rigid in Curry.

Exported constructors:

- `Rigid :: CaseType`
- `Flex :: CaseType`

`data CombType`

Data type for classifying combinations (i.e., a function/constructor applied to some arguments).

Exported constructors:

- `FuncCall :: CombType`
`FuncCall`
 - a call to a function where all arguments are provided
- `ConsCall :: CombType`
`ConsCall`
 - a call with a constructor at the top, all arguments are provided
- `FuncPartCall :: Int → CombType`
`FuncPartCall`
 - a partial call to a function (i.e., not all arguments are provided) where the parameter is the number of missing arguments
- `ConsPartCall :: Int → CombType`
`ConsPartCall`
 - a partial call to a constructor (i.e., not all arguments are provided) where the parameter is the number of missing arguments

`data Expr`

Data type for representing expressions.

Remarks:

if-then-else expressions are represented as function calls:

```
(if e1 then e2 else e3)
```

is represented as

```
(Comb FuncCall ("Prelude","if_then_else") [e1,e2,e3])
```

Higher-order applications are represented as calls to the (external) function `apply`. For instance, the rule

```
app f x = f x
```

is represented as

```
(Rule [0,1] (Comb FuncCall ("Prelude","apply") [Var 0, Var 1]))
```

A conditional rule is represented as a call to an external function `cond` where the first argument is the condition (a constraint). For instance, the rule

```
equal2 x | x:=2 = True
```

is represented as

```
(Rule [0]
  (Comb FuncCall ("Prelude","cond")
    [Comb FuncCall ("Prelude","=") [Var 0, Lit (Intc 2)],
     Comb FuncCall ("Prelude","True") []]))
```

Exported constructors:

- `Var :: Int → Expr`

`Var`

– variable (represented by unique index)

- `Lit :: Literal → Expr`

`Lit`

– literal (Int/Float/Char constant)

- `Comb :: CombType → (String,String) → [Expr] → Expr`
`Comb`
 – application (`f e1 ... en`) of function/constructor `f` with `n ≤ arity(f)`
- `Let :: [(Int,Expr)] → Expr → Expr`
`Let`
 – introduction of local variables via (recursive) let declarations
- `Free :: [Int] → Expr → Expr`
`Free`
 – introduction of free local variables
- `Or :: Expr → Expr → Expr`
`Or`
 – disjunction of two expressions (used to translate rules with overlapping left-hand sides)
- `Case :: CaseType → Expr → [BranchExpr] → Expr`
`Case`
 – case distinction (rigid or flex)
- `Typed :: Expr → TypeExpr → Expr`
`Typed`
 – typed expression to represent an expression with a type declaration

`data BranchExpr`

Data type for representing branches in a case expression.

Branches "`(m.c x1...xn) -> e`" in case expressions are represented as

```
(Branch (Pattern (m,c) [i1,...,in]) e)
```

where each `ij` is the index of the pattern variable `xj`, or as

```
(Branch (LPattern (Intc i)) e)
```

for integers as branch patterns (similarly for other literals like float or character constants).

Exported constructors:

- `Branch :: Pattern → Expr → BranchExpr`

`data Pattern`

Data type for representing patterns in case expressions.

Exported constructors:

- `Pattern :: (String,String) → [Int] → Pattern`
- `LPattern :: Literal → Pattern`

`data Literal`

Data type for representing literals occurring in an expression or case branch. It is either an integer, a float, or a character constant.

Exported constructors:

- `Intc :: Int → Literal`
- `Floatc :: Float → Literal`
- `Charc :: Char → Literal`

Exported functions:

`showQNameInModule :: String → (String,String) → String`

Translates a given qualified type name into external name relative to a module. Thus, names not defined in this module (except for names defined in the prelude) are prefixed with their module name.

A.5.7 Library FlatCurry.Files

This library supports meta-programming, i.e., the manipulation of Curry programs in Curry. This library defines I/O actions

Exported functions:

`readFlatCurry :: String → IO Prog`

I/O action which parses a Curry program and returns the corresponding FlatCurry program. Thus, the argument is the module path (without suffix `".curry"` or `".lcurry"`) and the result is a FlatCurry term representing this program.

`readFlatCurryWithParseOptions :: String → FrontendParams → IO Prog`

I/O action which parses a Curry program with respect to some parser options and returns the corresponding FlatCurry program. This I/O action is used by the standard action `readFlatCurry`.

`flatCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix `".curry"` or `".lcurry"`) into the name of the file containing the corresponding FlatCurry program.

`flatCurryIntName :: String → String`

Transforms a name of a Curry program (with or without suffix `".curry"` or `".lcurry"`) into the name of the file containing the corresponding FlatCurry program.

`readFlatCurryFile :: String → IO Prog`

I/O action which reads a FlatCurry program from a file in `".fcy"` format. In contrast to `readFlatCurry`, this action does not parse a source program. Thus, the argument must be the name of an existing file (with suffix `".fcy"`) containing a FlatCurry program in `".fcy"` format and the result is a FlatCurry term representing this program.

`readFlatCurryInt :: String → IO Prog`

I/O action which returns the interface of a Curry module, i.e., a FlatCurry program containing only `"Public"` entities and function definitions without rules (i.e., external functions). The argument is the file name without suffix `".curry"` (or `".lcurry"`) and the result is a FlatCurry term representing the interface of this module.

`writeFCY :: String → Prog → IO ()`

Writes a FlatCurry program into a file in `".fcy"` format. The first argument must be the name of the target file (with suffix `".fcy"`).

`lookupFlatCurryFileInLoadPath :: String → IO (Maybe String)`

Returns the name of the FlatCurry file of a module in the load path, if this file exists.

`getFlatCurryFileInLoadPath :: String → IO String`

Returns the name of the FlatCurry file of a module in the load path, if this file exists.

A.5.8 Library FlatCurry.Goodies

This library provides selector functions, test and update operations as well as some useful auxiliary functions for FlatCurry data terms. Most of the provided functions are based on general transformation functions that replace constructors with user-defined functions. For recursive datatypes the transformations are defined inductively over the term structure. This is quite usual for transformations on FlatCurry terms, so the provided functions can be used to implement specific transformations without having to explicitly state the recursion. Essentially, the tedious part of such transformations - descend in fairly complex term structures - is abstracted away, which hopefully makes the code more clear and brief.

Exported types:

`type Update a b = (b → b) → a → a`

Exported functions:

`trProg :: (String → [String] → [TypeDecl] → [FuncDecl] → [OpDecl] → a) → Prog → a`

transform program

`progName :: Prog → String`

get name from program

`progImports :: Prog → [String]`

get imports from program

`progTypes :: Prog → [TypeDecl]`

get type declarations from program

`progFuncs :: Prog → [FuncDecl]`

get functions from program

`progOps :: Prog → [OpDecl]`

get infix operators from program

`updProg :: (String → String) → ([String] → [String]) → ([TypeDecl] → [TypeDecl]) → ([FuncDecl] → [FuncDecl]) → ([OpDecl] → [OpDecl]) → Prog → Prog`

update program

`updProgName :: (String → String) → Prog → Prog`

update name of program

`updProgImports :: ([String] → [String]) → Prog → Prog`

update imports of program

`updProgTypes :: ([TypeDecl] → [TypeDecl]) → Prog → Prog`

update type declarations of program

`updProgFuncs :: ([FuncDecl] → [FuncDecl]) → Prog → Prog`

update functions of program

`updProgOps :: ([OpDecl] → [OpDecl]) → Prog → Prog`

update infix operators of program

`allVarsInProg :: Prog → [Int]`

get all program variables (also from patterns)

```
updProgExps :: (Expr → Expr) → Prog → Prog
```

lift transformation on expressions to program

```
rnmAllVarsInProg :: (Int → Int) → Prog → Prog
```

rename programs variables

```
updQNamesInProg :: ((String,String) → (String,String)) → Prog → Prog
```

update all qualified names in program

```
rnmProg :: String → Prog → Prog
```

rename program (update name of and all qualified names in program)

```
trType :: ((String,String) → Visibility → [Int] → [ConsDecl] → a) →
((String,String) → Visibility → [Int] → TypeExpr → a) → TypeDecl → a
```

transform type declaration

```
typeName :: TypeDecl → (String,String)
```

get name of type declaration

```
typeVisibility :: TypeDecl → Visibility
```

get visibility of type declaration

```
typeParams :: TypeDecl → [Int]
```

get type parameters of type declaration

```
typeConsDecls :: TypeDecl → [ConsDecl]
```

get constructor declarations from type declaration

```
typeSyn :: TypeDecl → TypeExpr
```

get synonym of type declaration

```
isTypeSyn :: TypeDecl → Bool
```

is type declaration a type synonym?

```
updType :: ((String,String) → (String,String)) → (Visibility → Visibility)
→ ([Int] → [Int]) → ([ConsDecl] → [ConsDecl]) → (TypeExpr → TypeExpr) →
TypeDecl → TypeDecl
```

update type declaration

```
updTypeName :: ((String,String) → (String,String)) → TypeDecl → TypeDecl
```


update name of type declaration

```
updTypeVisibility :: (Visibility → Visibility) → TypeDecl → TypeDecl
```

update visibility of type declaration

```
updTypeParams :: ([Int] → [Int]) → TypeDecl → TypeDecl
```

update type parameters of type declaration

```
updTypeConsDecls :: ([ConsDecl] → [ConsDecl]) → TypeDecl → TypeDecl
```

update constructor declarations of type declaration

```
updTypeSynonym :: (TypeExpr → TypeExpr) → TypeDecl → TypeDecl
```

update synonym of type declaration

```
updQNamesInType :: ((String,String) → (String,String)) → TypeDecl → TypeDecl
```

update all qualified names in type declaration

```
trCons :: ((String,String) → Int → Visibility → [TypeExpr] → a) → ConsDecl → a
```

transform constructor declaration

```
consName :: ConsDecl → (String,String)
```

get name of constructor declaration

```
consAry :: ConsDecl → Int
```

get arity of constructor declaration

```
consVisibility :: ConsDecl → Visibility
```

get visibility of constructor declaration

```
consArgs :: ConsDecl → [TypeExpr]
```

get arguments of constructor declaration

```
updCons :: ((String,String) → (String,String)) → (Int → Int) → (Visibility → Visibility) → ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
```

update constructor declaration

```
updConsName :: ((String,String) → (String,String)) → ConsDecl → ConsDecl
```

update name of constructor declaration

```
updConsAry :: (Int → Int) → ConsDecl → ConsDecl
```

update arity of constructor declaration

```

updConsVisibility :: (Visibility → Visibility) → ConsDecl → ConsDecl
    update visibility of constructor declaration

updConsArgs :: ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
    update arguments of constructor declaration

updQNamesInConsDecl :: ((String,String) → (String,String)) → ConsDecl →
ConsDecl
    update all qualified names in constructor declaration

tVarIndex :: TypeExpr → Int
    get index from type variable

domain :: TypeExpr → TypeExpr
    get domain from functional type

range :: TypeExpr → TypeExpr
    get range from functional type

tConsName :: TypeExpr → (String,String)
    get name from constructed type

tConsArgs :: TypeExpr → [TypeExpr]
    get arguments from constructed type

trTypeExpr :: (Int → a) → ((String,String) → [a] → a) → (a → a → a) →
TypeExpr → a
    transform type expression

isTVar :: TypeExpr → Bool
    is type expression a type variable?

isTCons :: TypeExpr → Bool
    is type declaration a constructed type?

isFuncType :: TypeExpr → Bool
    is type declaration a functional type?

updTVars :: (Int → TypeExpr) → TypeExpr → TypeExpr
    update all type variables

updTCons :: ((String,String) → [TypeExpr] → TypeExpr) → TypeExpr → TypeExpr

```

update all type constructors

`updFuncTypes :: (TypeExpr → TypeExpr → TypeExpr) → TypeExpr → TypeExpr`
 update all functional types

`argTypes :: TypeExpr → [TypeExpr]`
 get argument types from functional type

`resultType :: TypeExpr → TypeExpr`
 get result type from (nested) functional type

`renmAllVarsInTypeExpr :: (Int → Int) → TypeExpr → TypeExpr`
 rename variables in type expression

`updQNamesInTypeExpr :: ((String,String) → (String,String)) → TypeExpr → TypeExpr`
 update all qualified names in type expression

`trOp :: ((String,String) → Fixity → Int → a) → OpDecl → a`
 transform operator declaration

`opName :: OpDecl → (String,String)`
 get name from operator declaration

`opFixity :: OpDecl → Fixity`
 get fixity of operator declaration

`opPrecedence :: OpDecl → Int`
 get precedence of operator declaration

`updOp :: ((String,String) → (String,String)) → (Fixity → Fixity) → (Int → Int) → OpDecl → OpDecl`
 update operator declaration

`updOpName :: ((String,String) → (String,String)) → OpDecl → OpDecl`
 update name of operator declaration

`updOpFixity :: (Fixity → Fixity) → OpDecl → OpDecl`
 update fixity of operator declaration

`updOpPrecedence :: (Int → Int) → OpDecl → OpDecl`
 update precedence of operator declaration

```

trFunc :: ((String,String) → Int → Visibility → TypeExpr → Rule → a) →
FuncDecl → a
    transform function

funcName :: FuncDecl → (String,String)
    get name of function

funcArity :: FuncDecl → Int
    get arity of function

funcVisibility :: FuncDecl → Visibility
    get visibility of function

funcType :: FuncDecl → TypeExpr
    get type of function

funcRule :: FuncDecl → Rule
    get rule of function

updFunc :: ((String,String) → (String,String)) → (Int → Int) → (Visibility →
Visibility) → (TypeExpr → TypeExpr) → (Rule → Rule) → FuncDecl → FuncDecl
    update function

updFuncName :: ((String,String) → (String,String)) → FuncDecl → FuncDecl
    update name of function

updFuncArity :: (Int → Int) → FuncDecl → FuncDecl
    update arity of function

updFuncVisibility :: (Visibility → Visibility) → FuncDecl → FuncDecl
    update visibility of function

updFuncType :: (TypeExpr → TypeExpr) → FuncDecl → FuncDecl
    update type of function

updFuncRule :: (Rule → Rule) → FuncDecl → FuncDecl
    update rule of function

isExternal :: FuncDecl → Bool
    is function externally defined?

allVarsInFunc :: FuncDecl → [Int]

```

```

    get variable names in a function declaration
funcArgs :: FuncDecl → [Int]
    get arguments of function, if not externally defined
funcBody :: FuncDecl → Expr
    get body of function, if not externally defined
funcRHS :: FuncDecl → [Expr]

rnmAllVarsInFunc :: (Int → Int) → FuncDecl → FuncDecl
    rename all variables in function
updQNamesInFunc :: ((String,String) → (String,String)) → FuncDecl → FuncDecl
    update all qualified names in function
updFuncArgs :: ([Int] → [Int]) → FuncDecl → FuncDecl
    update arguments of function, if not externally defined
updFuncBody :: (Expr → Expr) → FuncDecl → FuncDecl
    update body of function, if not externally defined
trRule :: ([Int] → Expr → a) → (String → a) → Rule → a
    transform rule
ruleArgs :: Rule → [Int]
    get rules arguments if it's not external
ruleBody :: Rule → Expr
    get rules body if it's not external
ruleExtDecl :: Rule → String
    get rules external declaration
isRuleExternal :: Rule → Bool
    is rule external?
updRule :: ([Int] → [Int]) → (Expr → Expr) → (String → String) → Rule → Rule
    update rule
updRuleArgs :: ([Int] → [Int]) → Rule → Rule

```

update rules arguments

```
updRuleBody :: (Expr → Expr) → Rule → Rule
```

update rules body

```
updRuleExtDecl :: (String → String) → Rule → Rule
```

update rules external declaration

```
allVarsInRule :: Rule → [Int]
```

get variable names in a functions rule

```
rnmAllVarsInRule :: (Int → Int) → Rule → Rule
```

rename all variables in rule

```
updQNamesInRule :: ((String,String) → (String,String)) → Rule → Rule
```

update all qualified names in rule

```
trCombType :: a → (Int → a) → a → (Int → a) → CombType → a
```

transform combination type

```
isCombTypeFuncCall :: CombType → Bool
```

is type of combination FuncCall?

```
isCombTypeFuncPartCall :: CombType → Bool
```

is type of combination FuncPartCall?

```
isCombTypeConsCall :: CombType → Bool
```

is type of combination ConsCall?

```
isCombTypeConsPartCall :: CombType → Bool
```

is type of combination ConsPartCall?

```
missingArgs :: CombType → Int
```

varNr :: Expr → Int

get internal number of variable

```
literal :: Expr → Literal
```

get literal if expression is literal expression

```
combType :: Expr → CombType
```

get combination type of a combined expression

```
combName :: Expr → (String,String)
```

get name of a combined expression

```
combArgs :: Expr → [Expr]
```

get arguments of a combined expression

```
missingCombArgs :: Expr → Int
```

get number of missing arguments if expression is combined

```
letBinds :: Expr → [(Int,Expr)]
```

get indices of variables in let declaration

```
letBody :: Expr → Expr
```

get body of let declaration

```
freeVars :: Expr → [Int]
```

get variable indices from declaration of free variables

```
freeExpr :: Expr → Expr
```

get expression from declaration of free variables

```
orExps :: Expr → [Expr]
```

get expressions from or-expression

```
caseType :: Expr → CaseType
```

get case-type of case expression

```
caseExpr :: Expr → Expr
```

get scrutinee of case expression

```
caseBranches :: Expr → [BranchExpr]
```

get branch expressions from case expression

```
isVar :: Expr → Bool
```

is expression a variable?

```
isLit :: Expr → Bool
```

is expression a literal expression?

```
isComb :: Expr → Bool
```

is expression combined?

`isLet :: Expr → Bool`

is expression a let expression?

`isFree :: Expr → Bool`

is expression a declaration of free variables?

`isOr :: Expr → Bool`

is expression an or-expression?

`isCase :: Expr → Bool`

is expression a case expression?

`trExpr :: (Int → a) → (Literal → a) → (CombType → (String,String) → [a] → a) → ([Int,a] → a → a) → ([Int] → a → a) → (a → a → a) → (CaseType → a → [b] → a) → (Pattern → a → b) → (a → TypeExpr → a) → Expr → a`

transform expression

`updVars :: (Int → Expr) → Expr → Expr`

update all variables in given expression

`updLiterals :: (Literal → Expr) → Expr → Expr`

update all literals in given expression

`updCombs :: (CombType → (String,String) → [Expr] → Expr) → Expr → Expr`

update all combined expressions in given expression

`updLets :: ([Int,Expr] → Expr → Expr) → Expr → Expr`

update all let expressions in given expression

`updFrees :: ([Int] → Expr → Expr) → Expr → Expr`

update all free declarations in given expression

`updOrs :: (Expr → Expr → Expr) → Expr → Expr`

update all or expressions in given expression

`updCases :: (CaseType → Expr → [BranchExpr] → Expr) → Expr → Expr`

update all case expressions in given expression

`updBranches :: (Pattern → Expr → BranchExpr) → Expr → Expr`

update all case branches in given expression

`updTypeds :: (Expr → TypeExpr → Expr) → Expr → Expr`
 update all typed expressions in given expression

`isFuncCall :: Expr → Bool`
 is expression a call of a function where all arguments are provided?

`isFuncPartCall :: Expr → Bool`
 is expression a partial function call?

`isConsCall :: Expr → Bool`
 is expression a call of a constructor?

`isConsPartCall :: Expr → Bool`
 is expression a partial constructor call?

`isGround :: Expr → Bool`
 is expression fully evaluated?

`allVars :: Expr → [Int]`
 get all variables (also pattern variables) in expression

`rnmAllVars :: (Int → Int) → Expr → Expr`
 rename all variables (also in patterns) in expression

`updQNames :: ((String,String) → (String,String)) → Expr → Expr`
 update all qualified names in expression

`trBranch :: (Pattern → Expr → a) → BranchExpr → a`
 transform branch expression

`branchPattern :: BranchExpr → Pattern`
 get pattern from branch expression

`branchExpr :: BranchExpr → Expr`
 get expression from branch expression

`updBranch :: (Pattern → Pattern) → (Expr → Expr) → BranchExpr → BranchExpr`
 update branch expression

`updBranchPattern :: (Pattern → Pattern) → BranchExpr → BranchExpr`
 update pattern of branch expression

```

updBranchExpr :: (Expr → Expr) → BranchExpr → BranchExpr
    update expression of branch expression

trPattern :: ((String,String) → [Int] → a) → (Literal → a) → Pattern → a
    transform pattern

patCons :: Pattern → (String,String)
    get name from constructor pattern

patArgs :: Pattern → [Int]
    get arguments from constructor pattern

patLiteral :: Pattern → Literal
    get literal from literal pattern

isConsPattern :: Pattern → Bool
    is pattern a constructor pattern?

updPattern :: ((String,String) → (String,String)) → ([Int] → [Int]) → (Literal
→ Literal) → Pattern → Pattern
    update pattern

updPatCons :: ((String,String) → (String,String)) → Pattern → Pattern
    update constructors name of pattern

updPatArgs :: ([Int] → [Int]) → Pattern → Pattern
    update arguments of constructor pattern

updPatLiteral :: (Literal → Literal) → Pattern → Pattern
    update literal of pattern

patExpr :: Pattern → Expr
    build expression from pattern

```

A.5.9 Library FlatCurry.Pretty

This library provides pretty-printers for FlatCurry modules and all substructures (e.g., expressions).

Exported types:

`data Options`

Options for pretty printing

Exported constructors:

- `Options :: Int → QualMode → String → Options`

`data QualMode`

Qualification mode, determines whether identifiers are printed qualified or unqualified. While `QualNone` and `QualImports` aim at readability, there may be ambiguities due to shadowing. On the contrary, `QualImports` and `QualAll` produce correct output at the cost of readability.

Exported constructors:

- `QualNone :: QualMode`
`QualNone`
 - no qualification, only unqualified names
- `QualImportsButPrelude :: QualMode`
`QualImportsButPrelude`
 - qualify all imports except those from the module `Prelude`
- `QualImports :: QualMode`
`QualImports`
 - qualify all imports, including `Prelude`
- `QualAll :: QualMode`
`QualAll`
 - qualify all names

Exported functions:

`indentWidth :: Options → Int`

`qualMode :: Options → QualMode`

`currentModule :: Options → String`

`defaultOptions :: Options`

Default Options for pretty-printing.

`ppProg :: Options → Prog → Doc`

pretty-print a FlatCurry module

`ppHeader :: Options → String → [TypeDecl] → [FuncDecl] → Doc`

pretty-print the module header

`ppExports :: Options → [TypeDecl] → [FuncDecl] → Doc`

pretty-print the export list

`ppTypeExport :: Options → TypeDecl → Doc`

pretty-print a type export

`ppConsExports :: Options → [ConsDecl] → [Doc]`

pretty-print the export list of constructors

`ppFuncExports :: Options → [FuncDecl] → [Doc]`

pretty-print the export list of functions

`ppImports :: Options → [String] → Doc`

pretty-print a list of import statements

`ppImport :: Options → String → Doc`

pretty-print a single import statement

`ppOpDecls :: Options → [OpDecl] → Doc`

pretty-print a list of operator fixity declarations

`ppOpDecl :: Options → OpDecl → Doc`

pretty-print a single operator fixity declaration

`ppFixity :: Fixity → Doc`

pretty-print the associativity keyword

`ppTypeDecls :: Options → [TypeDecl] → Doc`

pretty-print a list of type declarations

`ppTypeDecl :: Options → TypeDecl → Doc`

pretty-print a type declaration

```
ppConsDecls :: Options → [ConsDecl] → Doc
```

pretty-print the constructor declarations

```
ppConsDecl :: Options → ConsDecl → Doc
```

pretty print a single constructor

```
ppTypeExp :: Options → TypeExpr → Doc
```

pretty a top-level type expression

```
ppTypeExpr :: Options → Int → TypeExpr → Doc
```

pretty-print a type expression

```
ppTVarIndex :: Int → Doc
```

pretty-print a type variable

```
ppFuncDecls :: Options → [FuncDecl] → Doc
```

pretty-print a list of function declarations

```
ppFuncDecl :: Options → FuncDecl → Doc
```

pretty-print a function declaration

```
ppRule :: Options → Rule → Doc
```

pretty-print a function rule

```
ppExp :: Options → Expr → Doc
```

Pretty-print a top-level expression.

```
ppExpr :: Options → Int → Expr → Doc
```

pretty-print an expression

```
ppVarIndex :: Int → Doc
```

pretty-print a variable

```
ppLiteral :: Literal → Doc
```

pretty-print a literal

```
ppComb :: Options → Int → (String,String) → [Expr] → Doc
```

Pretty print a constructor or function call

```
ppDecls :: Options → [(Int,Expr)] → Doc
```

pretty-print a list of declarations

```
ppDecl :: Options → (Int,Expr) → Doc
```

pretty-print a single declaration

```
ppCaseType :: CaseType → Doc
```

Pretty print the type of a case expression

```
ppBranch :: Options → BranchExpr → Doc
```

Pretty print a case branch

```
ppPattern :: Options → Pattern → Doc
```

Pretty print a pattern

```
ppPrefixQOp :: Options → (String,String) → Doc
```

pretty-print a qualified prefix operator.

```
ppPrefixOp :: (String,String) → Doc
```

pretty-print a prefix operator unqualified.

```
ppInfixQOp :: Options → (String,String) → Doc
```

pretty-print an infix operator

```
ppQName :: Options → (String,String) → Doc
```

Pretty-print a qualified name

```
ppName :: (String,String) → Doc
```

Pretty-print a qualified name unqualified (e.g., for type definitions).

```
isInfixOp :: (String,String) → Bool
```

Check whether an operator is an infix operator

```
isConsId :: (String,String) → Bool
```

Check whether an identifier represents the `:` list constructor.

```
isListId :: (String,String) → Bool
```

Check whether an identifier represents a list

```
isTupleId :: (String,String) → Bool
```

Check whether an identifier represents a tuple

```
indent :: Options → Doc → Doc
```

Indentation

A.5.10 Library FlatCurry.Read

This library defines operations to read a FlatCurry programs or interfaces together with all its imported modules in the current load path.

Exported functions:

`readFlatCurryWithImports :: String → IO [Prog]`

Reads a FlatCurry program together with all its imported modules. The argument is the name of the main module, possibly with a directory prefix.

`readFlatCurryWithImportsInPath :: [String] → String → IO [Prog]`

Reads a FlatCurry program together with all its imported modules in a given load path. The arguments are a load path and the name of the main module.

`readFlatCurryIntWithImports :: String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces. The argument is the name of the main module, possibly with a directory prefix. If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

`readFlatCurryIntWithImportsInPath :: [String] → String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces in a given load path. The arguments are a load path and the name of the main module. If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

A.5.11 Library FlatCurry.Show

This library contains operations to transform FlatCurry programs into string representations, either in a FlatCurry format or in a Curry-like syntax.

This library contains

- show functions for a string representation of FlatCurry programs (`showFlatProg`, `showFlatType`, `showFlatFunc`)
- functions for showing FlatCurry (type) expressions in (almost) Curry syntax (`showCurryType`, `showCurryExpr`,...).

Exported functions:

`showFlatProg :: Prog → String`

Shows a FlatCurry program term as a string (with some pretty printing).

`showFlatType :: TypeDecl → String`

`showFlatFunc :: FuncDecl → String`

`showCurryType :: ((String,String) → String) → Bool → TypeExpr → String`

Shows a FlatCurry type in Curry syntax.

`showCurryExpr :: ((String,String) → String) → Bool → Int → Expr → String`

Shows a FlatCurry expressions in (almost) Curry syntax.

`showCurryVar :: a → String`

`showCurryId :: String → String`

Shows an identifier in Curry form. Thus, operators are enclosed in brackets.

A.5.12 Library FlatCurry.XML

This library contains functions to convert FlatCurry programs into corresponding XML expressions and vice versa. This can be used to store Curry programs in a way independent of a Curry system or to use a Curry system, like PAKCS, as back end by other functional logic programming systems.

Exported functions:

`flatCurry2XmlFile :: Prog → String → IO ()`

Transforms a FlatCurry program term into a corresponding XML file.

`flatCurry2Xml :: Prog → XmlExp`

Transforms a FlatCurry program term into a corresponding XML expression.

`xmlFile2FlatCurry :: String → IO Prog`

Reads an XML file with a FlatCurry program and returns the FlatCurry program.

`xml2FlatCurry :: XmlExp → Prog`

Transforms an XML term into a FlatCurry program.

A.5.13 Library FlatCurry.FlexRigid

This library provides a function to compute the rigid/flex status of a FlatCurry expression (right-hand side of a function definition).

Exported types:

`data FlexRigidResult`

Datatype for representing a flex/rigid status of an expression.

Exported constructors:

- `UnknownFR :: FlexRigidResult`
- `ConflictFR :: FlexRigidResult`
- `KnownFlex :: FlexRigidResult`
- `KnownRigid :: FlexRigidResult`

Exported functions:

`getFlexRigid :: Expr → FlexRigidResult`

Computes the rigid/flex status of a FlatCurry expression. This function checks all cases in this expression. If the expression has rigid as well as flex cases (which cannot be the case for source level programs but might occur after some program transformations), the result `ConflictFR` is returned.

A.5.14 Library `FlatCurry.Compact`

This module contains functions to reduce the size of FlatCurry programs by combining the main module and all imports into a single program that contains only the functions directly or indirectly called from a set of main functions.

Exported types:

`data Option`

Options to guide the compactification process.

Exported constructors:

- `Verbose :: Option`
`Verbose`
 - for more output
- `Main :: String → Option`
`Main`
 - optimize for one main (unqualified!) function supplied here
- `Exports :: Option`
`Exports`

- optimize w.r.t. the exported functions of the module only
- `InitFuncs :: [(String,String)] → Option`
`InitFuncs`
 - optimize w.r.t. given list of initially required functions
- `Required :: [RequiredSpec] → Option`
`Required`
 - list of functions that are implicitly required and, thus, should not be deleted if the corresponding module is imported
- `Import :: String → Option`
`Import`
 - module that should always be imported (useful in combination with option `InitFuncs`)

`data RequiredSpec`

Data type to specify requirements of functions.

Exported constructors:

Exported functions:

`requires :: (String,String) → (String,String) → RequiredSpec`

(`fun requires reqfun`) specifies that the use of the function "fun" implies the application of function "reqfun".

`alwaysRequired :: (String,String) → RequiredSpec`

(`alwaysRequired fun`) specifies that the function "fun" should be always present if the corresponding module is loaded.

`defaultRequired :: [RequiredSpec]`

Functions that are implicitly required in a FlatCurry program (since they might be generated by external functions like "==" or "=:=" on the fly).

`generateCompactFlatCurryFile :: [Option] → String → String → IO ()`

Computes a single FlatCurry program containing all functions potentially called from a set of main functions and writes it into a FlatCurry file. This is done by merging all imported FlatCurry modules and removing the imported functions that are definitely not used.

`computeCompactFlatCurry :: [Option] → String → IO Prog`

Computes a single FlatCurry program containing all functions potentially called from a set of main functions. This is done by merging all imported FlatCurry modules (these are loaded demand-driven so that modules that contains no potentially called functions are not loaded) and removing the imported functions that are definitely not used.

A.5.15 Library FlatCurry.Annotated.Types

This library contains a version of FlatCurry's abstract syntax tree which can be annotated with arbitrary information due to a polymorphic type parameter. For instance, this could be used to annotate function declarations and expressions with their corresponding type.

For more information about the abstract syntax tree of FlatCurry, see the documentation of the respective module.

Exported types:

```
type Arity = Int
```

Arity of a function declaration

```
data AProg
```

Annotated FlatCurry program (corresponds to a module)

Exported constructors:

- `AProg :: String → [String] → [TypeDecl] → [AFuncDecl a] → [OpDecl] → AProg a`

```
data AFuncDecl
```

Annotated function declaration

Exported constructors:

- `AFunc :: (String,String) → Int → Visibility → TypeExpr → (ARule a) → AFuncDecl a`

```
data ARule
```

Annotated function rule

Exported constructors:

- `ARule :: a → [(Int,a)] → (AExpr a) → ARule a`
- `AExternal :: a → String → ARule a`

```
data AExpr
```

Annotated expression

Exported constructors:

- `AVar :: a → Int → AExpr a`
- `ALit :: a → Literal → AExpr a`

- `AComb :: a → CombType → ((String,String),a) → [AExpr a] → AExpr a`
- `ALet :: a → [((Int,a),AExpr a)] → (AExpr a) → AExpr a`
- `AFree :: a → [(Int,a)] → (AExpr a) → AExpr a`
- `AOr :: a → (AExpr a) → (AExpr a) → AExpr a`
- `ACase :: a → CaseType → (AExpr a) → [ABranchExpr a] → AExpr a`
- `ATyped :: a → (AExpr a) → TypeExpr → AExpr a`

`data ABranchExpr`

Annotated case branch

Exported constructors:

- `ABranch :: (APattern a) → (AExpr a) → ABranchExpr a`

`data APattern`

Annotated pattern

Exported constructors:

- `APattern :: a → ((String,String),a) → [(Int,a)] → APattern a`
- `ALPattern :: a → Literal → APattern a`

A.5.16 Library `FlatCurry.Annotated.Pretty`

This library provides pretty-printers for `AnnotatedFlatCurry` modules and all substructures (e.g., expressions). Note that annotations are ignored for pretty-printing.

Exported functions:

`ppProg :: AProg a → Doc`

pretty-print a `FlatCurry` module

`ppHeader :: String → [TypeDecl] → [AFuncDecl a] → Doc`

pretty-print the module header

`ppExports :: [TypeDecl] → [AFuncDecl a] → Doc`

pretty-print the export list

`ppTypeExport :: TypeDecl → Doc`

pretty-print a type export

`ppConsExports :: [ConsDecl] → [Doc]`

pretty-print the export list of constructors

```
ppFuncExports :: [AFuncDecl a] → [Doc]
```

pretty-print the export list of functions

```
ppImports :: [String] → Doc
```

pretty-print a list of import statements

```
ppImport :: String → Doc
```

pretty-print a single import statement

```
ppOpDecls :: [OpDecl] → Doc
```

pretty-print a list of operator fixity declarations

```
ppOpDecl :: OpDecl → Doc
```

pretty-print a single operator fixity declaration

```
ppFixity :: Fixity → Doc
```

pretty-print the associativity keyword

```
ppTypeDecls :: [TypeDecl] → Doc
```

pretty-print a list of type declarations

```
ppTypeDecl :: TypeDecl → Doc
```

pretty-print a type declaration

```
ppConsDecls :: [ConsDecl] → Doc
```

pretty-print the constructor declarations

```
ppConsDecl :: ConsDecl → Doc
```

pretty print a single constructor

```
ppTypeExp :: TypeExpr → Doc
```

pretty a top-level type expression

```
ppTypeExpr :: Int → TypeExpr → Doc
```

pretty-print a type expression

```
ppTVarIndex :: Int → Doc
```

pretty-print a type variable

```
ppFuncDecls :: [AFuncDecl a] → Doc
```

pretty-print a list of function declarations

```
ppFuncDecl :: AFuncDecl a → Doc
```

pretty-print a function declaration

```
ppRule :: ARule a → Doc
```

pretty-print a function rule

```
ppExp :: AExpr a → Doc
```

pretty-print a top-level expression

```
ppExpr :: Int → AExpr a → Doc
```

pretty-print an expression

```
ppAVarIndex :: (Int,a) → Doc
```

pretty-print an annotated variable

```
ppVarIndex :: Int → Doc
```

pretty-print a variable

```
ppLiteral :: Literal → Doc
```

pretty-print a literal

```
showEscape :: Char → String
```

Escape character literal

```
ppComb :: Int → ((String,String),a) → [AExpr b] → Doc
```

Pretty print a constructor or function call

```
ppDecls :: [((Int,a),AExpr b)] → Doc
```

pretty-print a list of declarations

```
ppDecl :: ((Int,a),AExpr b) → Doc
```

pretty-print a single declaration

```
ppCaseType :: CaseType → Doc
```

Pretty print the type of a case expression

```
ppBranch :: ABranchExpr a → Doc
```

Pretty print a case branch

```
ppPattern :: APattern a → Doc
```

Pretty print a pattern

`ppPrefixOp :: (String,String) → Doc`

pretty-print a prefix operator

`ppInfixOp :: (String,String) → Doc`

pretty-print an infix operator

`ppQName :: (String,String) → Doc`

Pretty-print a qualified name

`isInfixOp :: (String,String) → Bool`

Check whether an operator is an infix operator

`isListId :: (String,String) → Bool`

Check whether an identifier represents a list

`isTupleId :: (String,String) → Bool`

Check whether an identifier represents a tuple

`indent :: Doc → Doc`

Indentation

A.5.17 Library `FlatCurry.Annotated.Goodies`

This library provides selector functions, test and update operations as well as some useful auxiliary functions for `FlatCurry` data terms. Most of the provided functions are based on general transformation functions that replace constructors with user-defined functions. For recursive datatypes the transformations are defined inductively over the term structure. This is quite usual for transformations on `FlatCurry` terms, so the provided functions can be used to implement specific transformations without having to explicitly state the recursion. Essentially, the tedious part of such transformations - descend in fairly complex term structures - is abstracted away, which hopefully makes the code more clear and brief.

Exported types:

`type Update a b = (b → b) → a → a`

Exported functions:

`trProg :: (String → [String] → [TypeDecl] → [AFuncDecl a] → [OpDecl] → b) → AProg a → b`

transform program

`progName :: AProg a → String`

get name from program

`progImports :: AProg a → [String]`

get imports from program

`progTypes :: AProg a → [TypeDecl]`

get type declarations from program

`progFuncs :: AProg a → [AFuncDecl a]`

get functions from program

`progOps :: AProg a → [OpDecl]`

get infix operators from program

`updProg :: (String → String) → ([String] → [String]) → ([TypeDecl] → [TypeDecl]) → ([AFuncDecl a] → [AFuncDecl a]) → ([OpDecl] → [OpDecl]) → AProg a → AProg a`

update program

`updProgName :: (String → String) → AProg a → AProg a`

update name of program

`updProgImports :: ([String] → [String]) → AProg a → AProg a`

update imports of program

`updProgTypes :: ([TypeDecl] → [TypeDecl]) → AProg a → AProg a`

update type declarations of program

`updProgFuncs :: ([AFuncDecl a] → [AFuncDecl a]) → AProg a → AProg a`

update functions of program

`updProgOps :: ([OpDecl] → [OpDecl]) → AProg a → AProg a`

update infix operators of program

`allVarsInProg :: AProg a → [Int]`

get all program variables (also from patterns)

```
updProgExps :: (AExpr a → AExpr a) → AProg a → AProg a
```

lift transformation on expressions to program

```
rnmAllVarsInProg :: (Int → Int) → AProg a → AProg a
```

rename programs variables

```
updQNamesInProg :: ((String,String) → (String,String)) → AProg a → AProg a
```

update all qualified names in program

```
rnmProg :: String → AProg a → AProg a
```

rename program (update name of and all qualified names in program)

```
trType :: ((String,String) → Visibility → [Int] → [ConsDecl] → a) →
((String,String) → Visibility → [Int] → TypeExpr → a) → TypeDecl → a
```

transform type declaration

```
typeName :: TypeDecl → (String,String)
```

get name of type declaration

```
typeVisibility :: TypeDecl → Visibility
```

get visibility of type declaration

```
typeParams :: TypeDecl → [Int]
```

get type parameters of type declaration

```
typeConsDecls :: TypeDecl → [ConsDecl]
```

get constructor declarations from type declaration

```
typeSyn :: TypeDecl → TypeExpr
```

get synonym of type declaration

```
isTypeSyn :: TypeDecl → Bool
```

is type declaration a type synonym?

```
updType :: ((String,String) → (String,String)) → (Visibility → Visibility)
→ ([Int] → [Int]) → ([ConsDecl] → [ConsDecl]) → (TypeExpr → TypeExpr) →
TypeDecl → TypeDecl
```

update type declaration

```
updTypeName :: ((String,String) → (String,String)) → TypeDecl → TypeDecl
```

update name of type declaration

```
updTypeVisibility :: (Visibility → Visibility) → TypeDecl → TypeDecl
```

update visibility of type declaration

```
updTypeParams :: ([Int] → [Int]) → TypeDecl → TypeDecl
```

update type parameters of type declaration

```
updTypeConsDecls :: ([ConsDecl] → [ConsDecl]) → TypeDecl → TypeDecl
```

update constructor declarations of type declaration

```
updTypeSynonym :: (TypeExpr → TypeExpr) → TypeDecl → TypeDecl
```

update synonym of type declaration

```
updQNamesInType :: ((String,String) → (String,String)) → TypeDecl → TypeDecl
```

update all qualified names in type declaration

```
trCons :: ((String,String) → Int → Visibility → [TypeExpr] → a) → ConsDecl → a
```

transform constructor declaration

```
consName :: ConsDecl → (String,String)
```

get name of constructor declaration

```
consAry :: ConsDecl → Int
```

get arity of constructor declaration

```
consVisibility :: ConsDecl → Visibility
```

get visibility of constructor declaration

```
consArgs :: ConsDecl → [TypeExpr]
```

get arguments of constructor declaration

```
updCons :: ((String,String) → (String,String)) → (Int → Int) → (Visibility → Visibility) → ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
```

update constructor declaration

```
updConsName :: ((String,String) → (String,String)) → ConsDecl → ConsDecl
```

update name of constructor declaration

```
updConsAry :: (Int → Int) → ConsDecl → ConsDecl
```

update arity of constructor declaration

```

updConsVisibility :: (Visibility → Visibility) → ConsDecl → ConsDecl
    update visibility of constructor declaration

updConsArgs :: ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
    update arguments of constructor declaration

updQNamesInConsDecl :: ((String,String) → (String,String)) → ConsDecl →
ConsDecl
    update all qualified names in constructor declaration

tVarIndex :: TypeExpr → Int
    get index from type variable

domain :: TypeExpr → TypeExpr
    get domain from functional type

range :: TypeExpr → TypeExpr
    get range from functional type

tConsName :: TypeExpr → (String,String)
    get name from constructed type

tConsArgs :: TypeExpr → [TypeExpr]
    get arguments from constructed type

trTypeExpr :: (Int → a) → ((String,String) → [a] → a) → (a → a → a) →
TypeExpr → a

isTVar :: TypeExpr → Bool
    is type expression a type variable?

isTCons :: TypeExpr → Bool
    is type declaration a constructed type?

isFuncType :: TypeExpr → Bool
    is type declaration a functional type?

updTVars :: (Int → TypeExpr) → TypeExpr → TypeExpr
    update all type variables

updTCons :: ((String,String) → [TypeExpr] → TypeExpr) → TypeExpr → TypeExpr

```

update all type constructors

`updFuncTypes :: (TypeExpr → TypeExpr → TypeExpr) → TypeExpr → TypeExpr`
 update all functional types

`argTypes :: TypeExpr → [TypeExpr]`
 get argument types from functional type

`resultType :: TypeExpr → TypeExpr`
 get result type from (nested) functional type

`rnmAllVarsInTypeExpr :: (Int → Int) → TypeExpr → TypeExpr`
 rename variables in type expression

`updQNamesInTypeExpr :: ((String,String) → (String,String)) → TypeExpr → TypeExpr`
 update all qualified names in type expression

`trOp :: ((String,String) → Fixity → Int → a) → OpDecl → a`
 transform operator declaration

`opName :: OpDecl → (String,String)`
 get name from operator declaration

`opFixity :: OpDecl → Fixity`
 get fixity of operator declaration

`opPrecedence :: OpDecl → Int`
 get precedence of operator declaration

`updOp :: ((String,String) → (String,String)) → (Fixity → Fixity) → (Int → Int) → OpDecl → OpDecl`
 update operator declaration

`updOpName :: ((String,String) → (String,String)) → OpDecl → OpDecl`
 update name of operator declaration

`updOpFixity :: (Fixity → Fixity) → OpDecl → OpDecl`
 update fixity of operator declaration

`updOpPrecedence :: (Int → Int) → OpDecl → OpDecl`
 update precedence of operator declaration

```

trFunc :: ((String,String) → Int → Visibility → TypeExpr → ARule a → b) →
AFuncDecl a → b
    transform function

funcName :: AFuncDecl a → (String,String)
    get name of function

funcArity :: AFuncDecl a → Int
    get arity of function

funcVisibility :: AFuncDecl a → Visibility
    get visibility of function

funcType :: AFuncDecl a → TypeExpr
    get type of function

funcRule :: AFuncDecl a → ARule a
    get rule of function

updFunc :: ((String,String) → (String,String)) → (Int → Int) → (Visibility →
Visibility) → (TypeExpr → TypeExpr) → (ARule a → ARule a) → AFuncDecl a →
AFuncDecl a
    update function

updFuncName :: ((String,String) → (String,String)) → AFuncDecl a → AFuncDecl a
    update name of function

updFuncArity :: (Int → Int) → AFuncDecl a → AFuncDecl a
    update arity of function

updFuncVisibility :: (Visibility → Visibility) → AFuncDecl a → AFuncDecl a
    update visibility of function

updFuncType :: (TypeExpr → TypeExpr) → AFuncDecl a → AFuncDecl a
    update type of function

updFuncRule :: (ARule a → ARule a) → AFuncDecl a → AFuncDecl a
    update rule of function

isExternal :: AFuncDecl a → Bool
    is function externally defined?

```

```

allVarsInFunc :: AFuncDecl a → [Int]
    get variable names in a function declaration

funcArgs :: AFuncDecl a → [(Int,a)]
    get arguments of function, if not externally defined

funcBody :: AFuncDecl a → AExpr a
    get body of function, if not externally defined

funcRHS :: AFuncDecl a → [AExpr a]

rnmAllVarsInFunc :: (Int → Int) → AFuncDecl a → AFuncDecl a
    rename all variables in function

updQNamesInFunc :: ((String,String) → (String,String)) → AFuncDecl a →
AFuncDecl a
    update all qualified names in function

updFuncArgs :: ([(Int,a)] → [(Int,a)]) → AFuncDecl a → AFuncDecl a
    update arguments of function, if not externally defined

updFuncBody :: (AExpr a → AExpr a) → AFuncDecl a → AFuncDecl a
    update body of function, if not externally defined

trRule :: (a → [(Int,a)] → AExpr a → b) → (a → String → b) → ARule a → b
    transform rule

ruleArgs :: ARule a → [(Int,a)]
    get rules arguments if it's not external

ruleBody :: ARule a → AExpr a
    get rules body if it's not external

ruleExtDecl :: ARule a → String
    get rules external declaration

isRuleExternal :: ARule a → Bool
    is rule external?

updRule :: (a → a) → ([(Int,a)] → [(Int,a)]) → (AExpr a → AExpr a) → (String
→ String) → ARule a → ARule a

```

```

update rule
updRuleArgs :: ([[Int,a]] → [[Int,a]]) → ARule a → ARule a
    update rules arguments
updRuleBody :: (AExpr a → AExpr a) → ARule a → ARule a
    update rules body
updRuleExtDecl :: (String → String) → ARule a → ARule a
    update rules external declaration
allVarsInRule :: ARule a → [Int]
    get variable names in a functions rule
rnmAllVarsInRule :: (Int → Int) → ARule a → ARule a
    rename all variables in rule
updQNamesInRule :: ((String,String) → (String,String)) → ARule a → ARule a
    update all qualified names in rule
trCombType :: a → (Int → a) → a → (Int → a) → CombType → a
    transform combination type
isCombTypeFuncCall :: CombType → Bool
    is type of combination FuncCall?
isCombTypeFuncPartCall :: CombType → Bool
    is type of combination FuncPartCall?
isCombTypeConsCall :: CombType → Bool
    is type of combination ConsCall?
isCombTypeConsPartCall :: CombType → Bool
    is type of combination ConsPartCall?
missingArgs :: CombType → Int

varNr :: AExpr a → Int
    get internal number of variable
literal :: AExpr a → Literal

```

get literal if expression is literal expression
combType :: AExpr a → CombType
 get combination type of a combined expression
combName :: AExpr a → (String,String)
 get name of a combined expression
combArgs :: AExpr a → [AExpr a]
 get arguments of a combined expression
missingCombArgs :: AExpr a → Int
 get number of missing arguments if expression is combined
letBinds :: AExpr a → [(Int,a),AExpr a]
 get indices of variables in let declaration
letBody :: AExpr a → AExpr a
 get body of let declaration
freeVars :: AExpr a → [Int]
 get variable indices from declaration of free variables
freeExpr :: AExpr a → AExpr a
 get expression from declaration of free variables
orExps :: AExpr a → [AExpr a]
 get expressions from or-expression
caseType :: AExpr a → CaseType
 get case-type of case expression
caseExpr :: AExpr a → AExpr a
 get scrutinee of case expression
caseBranches :: AExpr a → [ABranchExpr a]

isVar :: AExpr a → Bool
 is expression a variable?

isLit :: AExpr a → Bool

is expression a literal expression?

`isComb :: AExpr a → Bool`

is expression combined?

`isLet :: AExpr a → Bool`

is expression a let expression?

`isFree :: AExpr a → Bool`

is expression a declaration of free variables?

`isOr :: AExpr a → Bool`

is expression an or-expression?

`isCase :: AExpr a → Bool`

is expression a case expression?

`trExpr :: (a → Int → b) → (a → Literal → b) → (a → CombType →
((String,String),a) → [b] → b) → (a → [((Int,a),b)] → b → b) → (a →
[(Int,a)] → b → b) → (a → b → b → b) → (a → CaseType → b → [c] → b)
→ (APattern a → b → c) → (a → b → TypeExpr → b) → AExpr a → b`

transform expression

`updVars :: (a → Int → AExpr a) → AExpr a → AExpr a`

update all variables in given expression

`updLiterals :: (a → Literal → AExpr a) → AExpr a → AExpr a`

update all literals in given expression

`updCombs :: (a → CombType → ((String,String),a) → [AExpr a] → AExpr a) →
AExpr a → AExpr a`

update all combined expressions in given expression

`updLets :: (a → [((Int,a),AExpr a)] → AExpr a → AExpr a) → AExpr a → AExpr a`

update all let expressions in given expression

`updFrees :: (a → [(Int,a)] → AExpr a → AExpr a) → AExpr a → AExpr a`

update all free declarations in given expression

`updOrs :: (a → AExpr a → AExpr a → AExpr a) → AExpr a → AExpr a`

update all or expressions in given expression

`updCases :: (a → CaseType → AExpr a → [ABranchExpr a] → AExpr a) → AExpr a → AExpr a`

update all case expressions in given expression

`updBranches :: (APattern a → AExpr a → ABranchExpr a) → AExpr a → AExpr a`

update all case branches in given expression

`updTypedes :: (a → AExpr a → TypeExpr → AExpr a) → AExpr a → AExpr a`

update all typed expressions in given expression

`isFunctionCall :: AExpr a → Bool`

is expression a call of a function where all arguments are provided?

`isFunctionPartCall :: AExpr a → Bool`

is expression a partial function call?

`isConsCall :: AExpr a → Bool`

is expression a call of a constructor?

`isConsPartCall :: AExpr a → Bool`

is expression a partial constructor call?

`isGround :: AExpr a → Bool`

is expression fully evaluated?

`allVars :: AExpr a → [Int]`

get all variables (also pattern variables) in expression

`rmAllVars :: (Int → Int) → AExpr a → AExpr a`

rename all variables (also in patterns) in expression

`updQNames :: ((String,String) → (String,String)) → AExpr a → AExpr a`

update all qualified names in expression

`trBranch :: (APattern a → AExpr a → b) → ABranchExpr a → b`

transform branch expression

`branchPattern :: ABranchExpr a → APattern a`

get pattern from branch expression

`branchExpr :: ABranchExpr a → AExpr a`

get expression from branch expression

`updBranch :: (APattern a → APattern a) → (AExpr a → AExpr a) → ABranchExpr a
→ ABranchExpr a`

update branch expression

`updBranchPattern :: (APattern a → APattern a) → ABranchExpr a → ABranchExpr a`

update pattern of branch expression

`updBranchExpr :: (AExpr a → AExpr a) → ABranchExpr a → ABranchExpr a`

update expression of branch expression

`trPattern :: (a → ((String,String),a) → [(Int,a)] → b) → (a → Literal → b)
→ APattern a → b`

transform pattern

`patCons :: APattern a → (String,String)`

get name from constructor pattern

`patArgs :: APattern a → [(Int,a)]`

get arguments from constructor pattern

`patLiteral :: APattern a → Literal`

get literal from literal pattern

`isConsPattern :: APattern a → Bool`

is pattern a constructor pattern?

`updPattern :: (((String,String),a) → ((String,String),a)) → ([(Int,a)] →
[(Int,a)]) → (Literal → Literal) → APattern a → APattern a`

update pattern

`updPatCons :: ((String,String) → (String,String)) → APattern a → APattern a`

update constructors name of pattern

`updPatArgs :: ([(Int,a)] → [(Int,a)]) → APattern a → APattern a`

update arguments of constructor pattern

`updPatLiteral :: (Literal → Literal) → APattern a → APattern a`

update literal of pattern

`patExpr :: APattern a → AExpr a`

build expression from pattern

`annRule :: ARule a → a`

`annExpr :: AExpr a → a`

Extract the annotation of an annotated expression.

`annPattern :: APattern a → a`

Extract the annotation of an annotated pattern.

`unAnnProg :: AProg a → Prog`

`unAnnFuncDecl :: AFuncDecl a → FuncDecl`

`unAnnRule :: ARule a → Rule`

`unAnnExpr :: AExpr a → Expr`

`unAnnPattern :: APattern a → Pattern`

A.5.18 Library `FlatCurry.Annotated.TypeSubst`

Type substitutions on type-annotated `AnnotatedFlatCurry`

Exported types:

`type AFCSubst = FM Int TypeExpr`

The (abstract) data type for substitutions on `TypeExpr`.

Exported functions:

`showAFCSubst :: FM Int TypeExpr → String`

`emptyAFCSubst :: FM Int TypeExpr`

The empty substitution

`lookupAFCSubst :: FM Int TypeExpr → Int → Maybe TypeExpr`

Searches the substitution for a mapping from the given variable index to a term.

`substFunc :: FM Int TypeExpr → AFuncDecl TypeExpr → AFuncDecl TypeExpr`

Applies a substitution to a function.

`substRule :: FM Int TypeExpr → ARule TypeExpr → ARule TypeExpr`

Applies a substitution to a type expression.

`substExpr :: FM Int TypeExpr → AExpr TypeExpr → AExpr TypeExpr`

Applies a substitution to a type expression.

`substSnd :: FM Int TypeExpr → (a, TypeExpr) → (a, TypeExpr)`

`substBranch :: FM Int TypeExpr → ABranchExpr TypeExpr → ABranchExpr TypeExpr`

Applies a substitution to a branch expression.

`substPattern :: FM Int TypeExpr → APattern TypeExpr → APattern TypeExpr`

Applies a substitution to a pattern.

`subst :: FM Int TypeExpr → TypeExpr → TypeExpr`

Looks up a type in a substitution and converts the resulting Term to a TypeExpr.
Returns a given default value if the lookup fails.

A.5.19 Library FlatCurry.Annotated.TypeInference

Library to annotate the expressions of a FlatCurry program with type information.

It can be used by any other Curry program which processes or transforms FlatCurry programs.
The main operation to use is

`inferProg :: Prog -> IO (Either String (AProg TypeExpr))`

which annotates a FlatCurry program with type information.

The type inference works in several steps:

1. For each known function and constructor, either imported or defined in the module itself, the respective type is inserted into a type environment (type assumption).
2. Every subexpression is annotated with a fresh type variable, whereas constructor and function names are annotated with a fresh variant of the type in the type assumption.
3. Based on FlatCurry's type inference rules, type equations are generated for a function's rule.
4. The resulting equations are solved using unification and the resulting substitution is applied to the function rule.
5. The inferred types are then normalized such that for every function rule the type variables start with 0.

In addition, the function `inferNewFunctions` allows to infer the types of a list of functions whose type is not known before. Consequently, this disallows polymorphic recursive functions. Those functions are separated into strongly connected components before their types are inferred to allow mutually recursive function definitions.

In case of any error, the type inference quits with an error message.

Exported types:

```
type TypeEnv = FM (String,String) TypeExpr
```

A type environment.

Exported functions:

```
inferProg :: Prog → IO (Either String (AProg TypeExpr))
```

Infers the type of a whole program.

```
inferProgFromProgEnv :: [(String,Prog)] → Prog → IO (Either String (AProg TypeExpr))
```

Infers the type of a whole program w.r.t. a list of imported modules.

```
inferFunction :: (String,String) → Prog → IO (Either String (AFuncDecl TypeExpr))
```

Infers the types of a single function specified by its qualified name.

```
inferNewFunctions :: Prog → [FuncDecl] → IO (Either String [AFuncDecl TypeExpr])
```

Infers the types of a group of (possibly mutually recursive) functions. Note that the functions are only monomorphically instantiated, i.e., polymorphic recursion is not supported. The given type may be too general, for instance a type variable only, and will be specialised to the inferred type.

```
inferExpr :: Prog → Expr → IO (Either String (AExpr TypeExpr))
```

```
inferProgEnv :: FM (String,String) TypeExpr → Prog → IO (Either String (AProg TypeExpr))
```

Infers the type of a whole program. Uses the given type environment instead of generating a new one.

```
inferFunctionEnv :: FM (String,String) TypeExpr → (String,String) → Prog → IO (Either String (AFuncDecl TypeExpr))
```

Infers the types of a single function specified by its qualified name. Uses the given type environment instead of generating a new one.

```
getTypeEnv :: Prog → IO (FM (String,String) TypeExpr)
```

Extract the type environment from the given Prog.

```
getTypeEnvFromProgEnv :: [(String,Prog)] → Prog → Either String (FM  
(String,String) TypeExpr)
```

Extract the type environment from the given Prog by lookup in a module name -> Prog environment.

A.5.20 Library CurryStringClassifier

The Curry string classifier is a simple tool to process strings containing Curry source code. The source string is classified into the following categories:

- moduleHead - module interface, imports, operators
- code - the part where the actual program is defined
- big comment - parts enclosed in {- ... -}
- small comment - from "--" to the end of a line
- text - a string, i.e. text enclosed in "..."
- letter - the given string is the representation of a character
- meta - containing information for meta programming

For an example to use the state scanner cf. addtypes, the tool to add function types to a given program.

Exported types:

```
type Tokens = [Token]
```

```
data Token
```

The different categories to classify the source code.

Exported constructors:

- SmallComment :: String → Token
- BigComment :: String → Token
- Text :: String → Token
- Letter :: String → Token
- Code :: String → Token
- ModuleHead :: String → Token
- Meta :: String → Token

Exported functions:

`isSmallComment :: Token → Bool`

test for category "SmallComment"

`isBigComment :: Token → Bool`

test for category "BigComment"

`isComment :: Token → Bool`

test if given token is a comment (big or small)

`isText :: Token → Bool`

test for category "Text" (String)

`isLetter :: Token → Bool`

test for category "Letter" (Char)

`isCode :: Token → Bool`

test for category "Code"

`isModuleHead :: Token → Bool`

test for category "ModuleHead", ie imports and operator declarations

`isMeta :: Token → Bool`

test for category "Meta", ie between {+ and +}

`scan :: String → [Token]`

Divides the given string into the six categories. For applications it is important to know whether a given part of code is at the beginning of a line or in the middle. The state scanner organizes the code in such a way that every string categorized as "Code" **always** starts in the middle of a line.

`plainCode :: [Token] → String`

Yields the program code without comments (but with the line breaks for small comments).

`unscan :: [Token] → String`

Inverse function of scan, i.e., $\text{unscan}(\text{scan } x) = x$. unscan is used to yield a program after changing the list of tokens.

`readScan :: String → IO [Token]`

return tokens for given filename

`testScan :: String → IO ()`

test whether (unscan . scan) is identity

B Markdown Syntax

This document describes the syntax of texts containing markdown elements. The markdown syntax is intended to simplify the writing of texts whose source is readable and can be easily formatted, e.g., as part of a web document. It is a subset of the [original markdown syntax](#) (basically, only internal links and pictures are missing) supported by the [Curry library Markdown](#).

B.1 Paragraphs and Basic Formatting

Paragraphs are separated by at least one line which is empty or does contain only blanks.

Inside a paragraph, one can *emphasize* text or also **strongly emphasize** text. This is done by wrapping it with one or two `_` or `*` characters:

```
_emphasize_  
*emphasize*  
__strong__  
**strong**
```

Furthermore, one can also mark `program code` text by backtick quotes (`'`):

```
The function 'fib' computes Fibonacci numbers.
```

Web links can be put in angle brackets, like in the link <http://www.google.com>:

```
<http://www.google.com>
```

Currently, only links starting with 'http' are recognized (so that one can also use HTML markup). If one wants to put a link under a text, one can put the text in square brackets directly followed by the link in round brackets, as in [Google](#):

```
[Google] (http://www.google.com)
```

If one wants to put a character that has a specific meaning in the syntax of Markdown, like `*` or `_`, in the output document, it should be escaped with a backslash, i.e., a backslash followed by a special character in the source text is translated into the given character (this also holds for program code, see below). For instance, the input text

```
\_word\_
```

produces the output `"_word_"`. The following backslash escapes are recognized:

```
\  backslash  
'  backtick  
*  asterisk  
_  underscore  
{ } curly braces  
[ ] square brackets
```

() parentheses
hash symbol
+ plus symbol
- minus symbol (dash)
. dot
blank
! exclamation mark

B.2 Lists and Block Formatting

An **unordered list** (i.e., without numbering) is introduced by putting a star in front of the list elements (where the star can be preceded by blanks). The individual list elements must contain the same indentation, as in

```
* First list element  
  with two lines
```

```
* Next list element.
```

```
  It contains two paragraphs.
```

```
* Final list element.
```

This is formatted as follows:

- First list element with two lines
- Next list element.
 It contains two paragraphs.
- Final list element.

Instead of a star, one can also put dashes or plus to mark unordered list items. Furthermore, one could nest lists. Thus, the input text

```
- Color:  
  + Yellow  
  + Read  
  + Blue  
- BW:  
  + Black  
  + White
```

is formatted as

- Color:

- Yellow
- Read
- Blue

- BW:

- Black
- White

Similarly, **ordered lists** (i.e., with numbering each item) are introduced by a number followed by a dot and at least one blank. All following lines belonging to the same numbered item must have the same indent as the first line. The actual value of the number is not important. Thus, the input

```
1. First element
```

```
99. Second
    element
```

is formatted as

1. First element
2. Second element

A quotation block is marked by putting a right angle followed by a blank in front of each line:

```
> This is
> a quotation.
```

It will be formatted as a quote element:

```
    This is a quotation.
```

A block containing **program code** starts with a blank line and is marked by intending each input line by *at least four spaces* where all following lines must have at least the same indentation as the first non-blank character of the first line:

```
    f x y = let z = (x,y)
              in (z,z)
```

The indentation is removed in the output:

```
f x y = let z = (x,y)
      in (z,z)
```

To visualize the structure of a document, one can also put a line containing only blanks and at least three dashes (stars would also work) in the source text:

```
-----
```

This is formatted as a horizontal line:



B.3 Headers

There are two forms to mark headers. In the first form, one can "underline" the main header in the source text by equal signs and the second-level header by dashes:

```
First-level header
=====
```

```
Second-level header
-----
```

Alternatively (and for more levels), one can prefix the header line by up to six hash characters, where the number of characters corresponds to the header level (where level 1 is the main header):

```
# Main header
```

```
## Level 2 header
```

```
### Level 3
```

```
#### Level 4
```

```
##### Level 5
```

```
##### Level 6
```



C Auxiliary Files

During the translation and execution of a Curry program with KiCS2, various intermediate representations of the source program are created and stored in different files which are shortly explained in this section. If you use KiCS2, it is not necessary to know about these auxiliary files because they are automatically generated and updated. You should only remember the command for deleting all auxiliary files (“`cleancurry`”, see Section 1.2) to clean up your directories.

The various components of KiCS2 create the following auxiliary files.

prog.fcy: This file contains the Curry program in the so-called “FlatCurry” representation where all functions are global (i.e., lambda lifting has been performed) and pattern matching is translated into explicit case/or expressions (compare Appendix A.1). This representation might be useful for other back ends and compilers for Curry and is the basis doing meta-programming in Curry. This file is implicitly generated when a program is compiled with KiCS2. It can be also explicitly generated by the Curry front end

```
cymake --flat -ikics2home/lib prog
```

The FlatCurry representation of a Curry program is usually generated by the front-end after parsing, type checking and eliminating local declarations.

If the Curry module M is stored in the directory dir , the corresponding FlatCurry program is stored in the directory “ $dir/.curry$ ”. This is also the case for hierarchical module names: if the module $D1.D2.M$ is stored in the directory dir (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding FlatCurry program is stored in “ $dir/.curry/D1/D2/M.fcy$ ”.

prog.fint: This file contains the interface of the program in the so-called “FlatCurry” representation, i.e., it is similar to `prog.fcy` but contains only exported entities and the bodies of all functions omitted (i.e., “external”). This representation is useful for providing a fast access to module interfaces. This file is implicitly generated when a program is compiled with KiCS2 and stored in the same directory as `prog.fcy`.

Curry_prog.hs: This file contains a Haskell program as the result of translating the Curry program with the KiCS2 compiler.

If the Curry module M is stored in the directory dir , the corresponding Haskell program is stored in the directory “ $dir/.curry/kics2$ ”. This is also the case for hierarchical module names: if the module $D1.D2.M$ is stored in the directory dir (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding Haskell program is stored in “ $dir/.curry/kics2/D1/D2/Curry_prog.hs$ ”.

Curry_prog.hi: This file contains the interface of the Haskell program `Curry_prog.hs` when the latter program is compiled in order to execute it. This file is stored in the same directory as `Curry_prog.hs`.

Curry_prog.o: This file contains the object code of the Haskell program `Curry_prog.hs` when the latter program is compiled in order to execute it. This file is stored in the same directory as `Curry_prog.hs`.

`Curry_prog.nda`: This file contains some information about the determinism behavior of operations that is used by the KiCS2 compiler (see [10] for more details about the use of this information). This file is stored in the same directory as `Curry_prog.hs`.

`Curry_prog.info`: This file contains some information about the top-level functions of module `prog` that are used by the interactive environment, like determinism behavior or IO status. This file is stored in the same directory as `Curry_prog.hs`.

`prog`: This file contains the executable after compiling and saving a program with KiCS2 (see command “:save” in Section 2.2).

D External Operations

Currently, KiCS2 has no general interface to external operations, i.e., operations whose semantics is not defined by program rules in a Curry program but by some code written in another programming language. Thus, if an external operation should be added to the system, this operation must be declared as `external` in the Curry source code and an implementation for this external operation must be provided for the run-time system. An external operation is defined as follows in the Curry source code:

1. Add a type declaration for the external operation somewhere in a module defining this operation (usually, the prelude or some system module).
2. For external operations it is not allowed to define any rule since their semantics is determined by an external implementation. Instead of the defining rules, you have to write

```
f external
```

below the type declaration for the external operation `f`.

Furthermore, an implementation of the external operation must be provided in the target language of the KiCS2 compiler, i.e., in Haskell, and inserted in the compiled code. In order to simplify this task, KiCS2 follows some code conventions that are described in the following.

Assume you want to implement your own concatenation for strings in a module `String`. The name and type of this string concatenation should be

```
sconc :: String → String → String
```

Since the primitive Haskell implementation of this operation does not now anything about the operational mechanism of Curry (e.g., needed narrowing, non-deterministic rewriting), the arguments need to be completely evaluated before the primitive implementation is called. This can be easily obtained by the prelude operation (`$$$`) that applies an operation to the *normal form* of the given argument, i.e., this operation evaluates the argument to its normal form before applying the operation to it.⁹ Thus, we define `sconc` by

```
sconc :: String → String → String
sconc s1 s2 = (prim_sconc $$$ s1) $$$ s2
```

```
prim_sconc :: String → String → String
prim_sconc external
```

so that it is ensured that the external operation `prim_sconc` is always called with complete evaluated arguments.

In order to define the Haskell code implementing `prim_sconc`, one has to satisfy the naming conventions of KiCS2. The KiCS2 compiler generates the following code for the external operation `prim_sconc` (note that the generated Haskell code for the module `String` is stored in the file `.curry/kics2/Curry_String.hs`):

⁹There is also a similar prelude operation (`$#`) which evaluates the argument only to head-normal form. This is a bit more efficient and can be used for unstructured types like `Bool`.

```

d_C_prim_sconc :: Curry_Prelude.OP_List Curry_Prelude.C_Char
               → Curry_Prelude.OP_List Curry_Prelude.C_Char
               → ConstStore
               → Curry_Prelude.OP_List Curry_Prelude.C_Char
d_C_prim_sconc x1 x2 x3500 = external_d_C_prim_sconc x1 x2 x3500

```

The type constructors `OP_List` and `C_Char` of the prelude `Curry_Prelude`¹⁰ correspond to the Curry type constructors for lists and characters. The Haskell operation `external_d_C_prim_sconc` is the external operation to be implemented in Haskell by the programmer. The additional argument of type `ConstStore` represents the current set of constraints when this operation is called. This argument is intended to provide a more efficient access to binding constraints and can be ignored in standard operations.

If `String.curry` contains the code of the Curry function `sconc` described above, the Haskell code implementing the external operations occurring in the module `String` must be in the file `External_String.hs` which is located in the same directory as the file `String.curry`. The KiCS2 compiler appends the code contained in `External_String.hs` to the generated code stored in the file `.curry/kics2/Curry_String.hs`.¹¹

In order to complete our example, we have to write into the file `External_String.hs` a definition of the Haskell function `external_d_C_prim_sconc`. Thus, we start with the following definitions:

```

import qualified Curry_Prelude as CP

external_d_C_prim_sconc :: CP.OP_List CP.C_Char → CP.OP_List CP.C_Char
                       → ConstStore → CP.OP_List CP.C_Char

```

First, we import the standard prelude with the name `CP` in order to shorten the writing of type declarations. In order to write the final code of this operation, we have to convert the Curry-related types (like `C_Char`) into the corresponding Haskell types (like `Char`). Note that the Curry-related types contain information about non-deterministic or constrained values (see [10, 9]) that are meaningless in Haskell. To solve this conversion problem, the implementation of KiCS2 provides a family of operations to perform these conversions for the predefined types occurring in the standard prelude. For instance, `fromCurry` converts a Curry type into the corresponding Haskell type, and `toCurry` converts the Haskell type into the corresponding Curry type. Thus, we complete our example with the definition (note that we simply ignore the final argument representing the constraint store)

```

external_d_C_prim_sconc s1 s2 _ =
  toCurry ((fromCurry s1 ++ fromCurry s2) :: String)

```

Here, we use Haskell’s concatenation operation “`++`” to concatenate the string arguments. The type annotation “`:: String`” is necessary because “`++`” is a polymorphic function so that the type inference system of Haskell has problems to determine the right instance of the conversion function.

The conversion between Curry types and Haskell types, i.e., the family of conversion operation `fromCurry` and `toCurry`, is defined in the KiCS2 implementation for all standard data types. In particular, it is also defined on function types so that one can easily implement external Curry I/O

¹⁰Note that all translated Curry modules are imported in the Haskell code fully qualified in order to avoid name conflicts.

¹¹If the file `External_String.hs` contains also some import declarations at the beginning, these import declarations are put after the generated import declarations.

actions by using Haskell I/O actions. For instance, if we want to implement an external operation to print some string as an output line, we start by declaring the external operations in the Curry module `String`:

```
printString :: String → IO ()
printString s = prim_printString $$$ s

prim_printString :: String → IO ()
prim_printString external
```

Next we add the corresponding implementation in the file `External_String.hs` (where `C_IO` and `OP_Unit` are the names of the Haskell representation of the Curry type constructor `IO` and the Curry data type “`()`”, respectively):

```
external_d_C_prim_printString :: CP.OP_List CP.C_Char → ConstStore
                                → CP.C_IO CP.OP_Unit
external_d_C_prim_printString s _ = toCurry putStrLn s
```

Here, Haskell’s I/O action `putStrLn` of type “`String -> IO ()`” is transformed into a Curry I/O action “`toCurry putStrLn`” which has the type

```
CP.OP_List CP.C_Char → CP.C_IO CP.OP_Unit
```

When we compile the Curry module `String`, `KiCS2` combines these definitions in the target program so that we can immediately use the externally defined operation `printString` in Curry programs.

As we have seen, `KiCS2` transforms a name like `primOP` of an external operation into the name `external_d_C_primOP` for the Haskell operation to be implemented, i.e., only a specific prefix is added. However, this is only valid if no special characters occur in the Curry names. Otherwise (in order to generate a correct Haskell program), special characters are translated into specific names prefixed by “`OP_`”. For instance, if we declare the external operation

```
(<&>) :: Int → Int → Int
(<&>) external
```

the generated Haskell module contains the code

```
d_OP_lt_ampersand_gt :: Curry_Prelude.C_Int → Curry_Prelude.C_Int
                      → ConstStore → Curry_Prelude.C_Int
d_OP_lt_ampersand_gt x1 x2 x3500 = external_d_OP_lt_ampersand_gt x1 x2 x3500
```

so that one has to implement the operation `external_d_OP_lt_ampersand_gt` in Haskell. If in doubt, one should look into the generated Haskell code about the names and types of the operations to be implemented.

Finally, note that this method to connect functions implemented in Haskell to Curry programs provides the opportunity to connect also operations written in other programming languages to Curry via Haskell’s foreign function interface.

Index

<, 106
***, 71
*., 69
+., 69
---, 28
-., 69
.kics2rc, 16
/., 69
//, 142
:!, 11
:&, 149
:add, 10
:browse, 11

addFormParam, 168
 addHeadings, 171
 addHours, 137
 addListToFM, 144
 addListToFM.C, 144
 addMinutes, 137
 addMonths, 137
 addPageParam, 169
 addRegionStyle, 84
 address, 171
 addSeconds, 137
 addSound, 168
 addToFM, 144
 addToFM.C, 144
 addTrailingPathSeparator, 66
 addVS, 141
 addYears, 137
 AExpr, 250
 AFCSbst, 267
 aFloat, 201
 AFuncDecl, 250
 aInt, 201
 align, 109
 allC, 53
 allDBInfos, 98
 allDBKeyInfos, 98
 allDBKeys, 98
 AllSolutions, 7
 allSolutions, 68
 allValues, 68
 allValuesBFS, 128, 139
 allValuesDFS, 128, 139
 allValuesDiag, 128
 allValuesIDS, 128, 139
 allValuesIDSwith, 128, 140
 allValuesWith, 128
 allVars, 240, 265
 allVarsInFunc, 235, 261
 allVarsInProg, 230, 255
 allVarsInRule, 237, 262
 alwaysRequired, 249
 anchor, 171
 andC, 53
 angles, 115
 annExpr, 267
 annPattern, 267
 annRule, 267
 answerEncText, 168
 answerText, 168
 anyC, 53
 APattern, 251
 appendStyledValue, 84
 appendValue, 84
 applyAt, 142
 applyE, 217
 applyF, 217
 applyJust, 217
 applyMaybe, 217
 applySubst, 161
 applyV, 217
 AProg, 250
 ArgDescr, 72
 ArgOrder, 72
 args, 15
 argTypes, 214, 234, 259
 Arity, 206, 250
 Array, 141
 ARule, 250
 as-pattern, 18
 asin, 70
 asinh, 70
 assert, 55
 assertEqual, 49
 assertEqualIO, 49
 assertIO, 49, 55
 Assertion, 49
 assertSolutions, 49
 assertTrue, 49
 assertValues, 49
 aString, 201
 at, 117
 atan, 70
 atanh, 70
 attr, 201
 backslash, 117
 bar, 117
 baseName, 63

baseType, 216
 best, 7
 bfs, 12
 bfsStrategy, 127, 139
 bgBlack, 119
 bgBlue, 120
 bgCyan, 120
 bgGreen, 119
 bgMagenta, 120
 bgRed, 119
 bgWhite, 120
 bgYellow, 120
 bindings, 13
 binomial, 87
 bitAnd, 87
 bitNot, 87
 bitOr, 87
 bitTrunc, 87
 bitXor, 87
 black, 119
 blink, 170
 blinkRapid, 118
 blinkSlow, 118
 block, 172
 blockstyle, 172
 blue, 119
 bold, 118, 170
 boolType, 216
 both, 71
 bquotes, 115
 braces, 115
 brackets, 115
 BranchExpr, 227
 branchExpr, 240, 265
 branchPattern, 240, 265
 breakline, 172
 buildGr, 149
 Button, 85
 button, 172

 CalendarTime, 135
 calendarTimeToString, 137
 callFrontend, 61
 callFrontendWithParams, 61

 CanvasItem, 80
 CanvasScroll, 85
 caseBranches, 238, 263
 caseExpr, 238, 263
 CaseType, 225
 caseType, 238, 263
 cat, 112
 categorizeByItemKey, 163
 catMaybes, 104
 cBranch, 217
 CCaseType, 211
 cChar, 218
 CConsDecl, 207
 center, 170
 CExpr, 210
 CField, 206
 CFieldDecl, 208
 CFixity, 208
 cfunc, 217
 CFuncDecl, 209
 CgiEnv, 163
 CgiRef, 163
 CgiServerMsg, 176
 cgiServerRegistry, 177
 char, 115, 200
 checkAssertion, 50
 checkbox, 173
 checkedbox, 173
 childFamilies, 159
 children, 159
 choices, 12
 choose, 131
 chooseColor, 86
 chooseValue, 132
 cleancurry, 6
 cleanDB, 99
 CLiteral, 211
 CLocalDecl, 209
 ClockTime, 135
 clockTimeToInt, 136
 closeDBHandles, 99
 Cmd, 85
 cmp, 14
 cmpChar, 157

- cmpList, 157
- cmpString, 157
- cmtfunc, 217
- code, 170
- col, 82
- colon, 116
- Color, 81
- ColVal, 95
- combArgs, 238, 263
- combine, 66, 110, 142
- combineSimilar, 142
- combName, 238, 263
- CombType, 225
- combType, 237, 263
- comma, 116
- Command, 85
- comment
 - documentation, 28
- compareCalendarTime, 137
- compareClockTime, 137
- compareDate, 137
- compose, 111
- computeCompactFlatCurry, 249
- concatMapES, 62
- concurrency, 7
- ConfCollection, 79
- ConfigButton, 85
- ConfItem, 75
- connectToCommand, 91
- connectToSocket, 106, 134
- connectToSocketRepeat, 105
- connectToSocketWait, 106
- cons, 143
- consArgs, 232, 257
- consArity, 232, 257
- ConsDecl, 223
- consName, 214, 232, 257
- constF, 217
- constructors, 213
- consVis, 214
- consVisibility, 232, 257
- Context, 148
- context, 150
- Context', 148
- cookieForm, 168
- CookieParam, 166
- coordinates, 175
- COpDecl, 208
- copyFile, 56
- cos, 70
- cosh, 70
- CPattern, 210
- cpnsAlive, 53
- cpnsShow, 53
- cpnsStart, 53
- cpnsStop, 53
- cpvar, 218
- createDirectory, 55
- createDirectoryIfMissing, 56
- CRhs, 209
- crossout, 119
- CRule, 209
- CStatement, 211
- ctDay, 136
- ctHour, 136
- ctMin, 136
- ctMonth, 136
- ctSec, 136
- ctTZ, 136
- ctvar, 218
- CTVarIName, 206
- ctYear, 136
- CTypeDecl, 207
- CTypeExpr, 208
- currentModule, 242
- Curry mode, 16
- Curry preprocessor, 35
- curryCompiler, 57
- curryCompilerMajorVersion, 57
- curryCompilerMinorVersion, 57
- CurryDoc, 28
- currydoc, 29
- CURRYPATH, 8
- CurryProg, 207
- curryRuntime, 57
- curryRuntimeMajorVersion, 58
- curryRuntimeMinorVersion, 58
- currySubdir, 59

- CurryTest, 33
- currytest, 33
- cvar, 218
- CVarIName, 206
- CVisibility, 206
- cyan, 119
- cycle, 103
- cyclic structure, 18
- cymake, 276

- database programming, 42
- dateType, 216
- daysOfMonth, 137
- debugTcl, 82
- Decomp, 148
- defaultEncoding, 167
- defaultOptions, 219, 243
- defaultParams, 59
- defaultRequired, 249
- deg, 151
- deg', 152
- delEdge, 149
- delEdges, 150
- delete, 100, 155
- deleteBy, 100
- deleteDBEntries, 99
- deleteDBEntry, 99
- deleteRBT, 156, 158
- delFromFM, 144
- dellistFromFM, 145
- delNode, 149
- delNodes, 150
- deqHead, 143
- deqInit, 143
- deqLast, 143
- deqLength, 143
- deqReverse, 143
- deqTail, 143
- deqToList, 143
- dfs, 12
- dfsStrategy, 127, 139
- diagStrategy, 128
- digitToInt, 51
- dirName, 63

- dlist, 171
- Doc, 107
- documentation comment, 28
- documentation generator, 28
- doesDirectoryExist, 55
- doesFileExist, 55
- domain, 233, 258
- doneT, 97
- dot, 116
- doubleArrow, 117
- doubleColon, 117
- dquote, 116
- dquotes, 115
- dropDrive, 65
- dropExtension, 65
- dropExtensions, 65
- dropFileName, 66
- dropTrailingPathSeparator, 66
- Dynamic, 95

- eBool, 202
- eChar, 202
- Edge, 147
- edges, 152
- eEmpty, 202
- eFloat, 202
- eInt, 201
- element, 200
- elemFM, 146
- elemIndex, 100
- elemIndices, 100
- elemRBT, 156
- elemsOf, 198
- eltsFM, 146
- Emacs, 16
- emap, 153
- emphasize, 170
- empty, 106, 107, 143, 149, 154, 200
- emptyAFCSbst, 267
- emptyDefaultArray, 142
- emptyErrorArray, 142
- emptyFM, 144
- emptySetRBT, 156
- emptySubst, 161

emptyTableRBT, 158
emptyVS, 141
encapsulated search, 7
enclose, 114
encloseSep, 113
encloseSepSpaced, 113
Encoding, 197
entity relationship diagrams, 42
EntryScroll, 85
eOpt, 202
eqFM, 145
equal, 151
equalFilePath, 67
equals, 117
ERD2Curry, 42
erd2curry, 42
eRep, 202
eRepSeq1, 203
eRepSeq2, 203
eRepSeq3, 204
eRepSeq4, 204
eRepSeq5, 205
eRepSeq6, 205
errorT, 97
ES, 62
eSeq1, 202
eSeq2, 203
eSeq3, 203
eSeq4, 204
eSeq5, 205
eSeq6, 205
eString, 202
evalChildFamilies, 159
evalChildFamiliesIO, 160
evalCmd, 91
evalES, 62
evalFamily, 159
evalFamilyIO, 160
even, 87
Event, 78
exclusiveIO, 91
execCmd, 91
existsDBKey, 98
exitGUI, 83
exitWith, 135
exp, 70
expires, 168
Expr, 225
extended, 60
extendSubst, 161
external operation, 278
extSeparator, 64
factorial, 86
failES, 62
failT, 97
failVS, 141
faint, 118
family, 159
FilePath, 64
fileSize, 55
fileSuffix, 63
fill, 117
fillBreak, 118
fillCat, 112
fillEncloseSep, 113
fillEncloseSepSpaced, 113
fillSep, 112
filterFM, 145
find, 100
findall, 7
findIndex, 100
findIndices, 100
first, 13, 71
fix, 71
Fixity, 224
FlatCurry, 47
flatCurry2Xml, 247
flatCurry2XmlFile, 247
flatCurryFileName, 228
flatCurryIntName, 229
FlexRigidResult, 248
float, 115, 200
floatType, 216
FM, 144
fmSortBy, 146
fmToList, 146
fmToListPreOrder, 146

focusInput, 84
 fold, 159
 foldChildren, 160
 foldFM, 145
 foldValues, 132
 footer, 170
 form, 167
 formatMarkdownFileAsPDF, 181
 formatMarkdownInputAsPDF, 181
 formBodyAttr, 167
 formCSS, 167
 formEnc, 167
 formMetaInfo, 167
 FormParam, 164
 freeExpr, 238, 263
 freeVars, 238, 263
 fromJust, 104
 fromLeft, 61
 fromMarkdownText, 180
 fromMaybe, 104
 fromRight, 61
 FrontendParams, 57
 FrontendTarget, 56
 fullPath, 60
 funcArgs, 236, 261
 funcArity, 235, 260
 funcBody, 236, 261
 FuncDecl, 224
 funcName, 215, 235, 260
 funcNamesOfFDecl, 215
 funcNamesOfLDecl, 215
 funcNamesOfStat, 215
 funcRHS, 236, 261
 funcRule, 235, 260
 functional pattern, 18
 functions, 213
 funcType, 235, 260
 funcVis, 215
 funcVisibility, 235, 260

 garbageCollect, 121
 garbageCollectorOff, 121
 garbageCollectorOn, 121
 GDecomp, 148

 gelem, 151
 generateCompactFlatCurryFile, 249
 germanLatexDoc, 176
 getAbsolutePath, 56
 getAllFailures, 48
 getAllSolutions, 48
 getAllValues, 48, 67
 getAllValuesWith, 128, 140
 getArgs, 134
 getAssoc, 92
 getClockTime, 136
 getContents, 90
 getContentsOfUrl, 181
 getCookies, 175
 getCPUTime, 134
 getCurrentDirectory, 55
 getCursorPosition, 84
 getDB, 97
 getDBInfo, 99
 getDBInfos, 99
 getDirectoryContents, 55
 getElapsedTime, 134
 getEnviron, 134
 getFileInPath, 63
 getFlatCurryFileInLoadPath, 229
 getFlexRigid, 248
 getHomeDirectory, 56
 getHostname, 134
 getLoadPathForModule, 59
 getLocalTime, 136
 getModificationTime, 55
 getOneSolution, 48
 getOneValue, 48
 getOpenFile, 85
 getOpenFileWithTypes, 85
 getOpt, 72
 getOpt', 72
 getPID, 135
 getPortInfo, 53
 getProcessInfos, 121
 getProgName, 135
 getRandomSeed, 154
 getRcVar, 58
 getRcVars, 58

- gets, 62
- getSaveFile, 86
- getSaveFileWithTypes, 86
- getSearchPath, 64
- getSearchTree, 127, 139
- getSomeValue, 68
- getTemporaryDirectory, 56
- getTypeEnv, 270
- getTypeEnvFromProgEnv, 270
- getUrlParameter, 174
- getValue, 83
- ghc, 12, 14
- ghci, 14
- Global, 73
- global, 73
- global installation, 6
- GlobalSpec, 73
- gmap, 152
- Graph, 148
- green, 119
- group, 101, 108
- groupBy, 101
- guardedRule, 217
- GuiPort, 74

- h1, 169
- h2, 169
- h3, 169
- h4, 170
- h5, 170
- Handle, 88
- hang, 108
- hardline, 108
- hasDrive, 65
- hasExtension, 65
- hasTrailingPathSeparator, 66
- hcat, 112
- hClose, 89
- headedTable, 171
- header, 170
- empty, 169
- hEncloseSep, 113
- hFlush, 89
- hGetChar, 90
- hGetContents, 90
- hGetLine, 90
- hiddenfield, 174
- hIsEOF, 89
- hIsReadable, 90
- hIsTerminalDevice, 90
- hIsWritable, 90
- hPrint, 90
- hPutChar, 90
- hPutStr, 90
- hPutStrLn, 90
- hReady, 90
- href, 171
- hrule, 171
- hSeek, 89
- hsep, 111
- htmldir, 60
- HtmlExp, 163
- HtmlForm, 164
- HtmlHandler, 163
- htmlIsoUmlauts, 174
- HtmlPage, 166
- htmlQuote, 174
- htmlSpecialChars2tex, 175
- htxt, 169
- htxts, 169
- hWaitForInput, 89
- hWaitForInputOrMsg, 89
- hWaitForInputs, 89
- hWaitForInputsOrMsg, 89

- i2f, 69
- identicalVars, 138
- idOfCgiRef, 167
- ids, 12
- idsStrategy, 127, 139
- idsStrategyWith, 128, 140
- ilog, 86
- image, 172
- imageButton, 172
- imports, 213
- inCurrySubdir, 59
- inCurrySubdirModule, 59
- indeg, 151

- indeg', 152
- indent, 109, 245, 254
- indentWidth, 242
- inferExpr, 269
- inferFunction, 269
- inferFunctionEnv, 269
- inferNewFunctions, 269
- inferProg, 269
- inferProgEnv, 269
- inferProgFromProgEnv, 269
- init, 102
- inits, 102
- inline, 172
- inn, 151
- inn', 152
- insEdge, 149
- insEdges, 150
- insertBy, 102
- insertMultiRBT, 156
- insertRBT, 156
- insNode, 149
- insNodes, 149
- installation
 - global, 6
 - local, 6
- installDir, 58
- int, 115, 200
- integer, 12
- interactive, 10
- interactive, 13
- intercalate, 101
- intersect, 101
- intersectBy, 101
- intersectFM, 145
- intersectFM_C, 145
- intersectRBT, 156
- intersperse, 101
- intForm, 176
- intFormMain, 176
- intToDigit, 51
- intType, 216
- inverse, 119
- IOMode, 88
- IORef, 91
- ioref, 13
- ioType, 216
- isAbsolute, 63, 67
- isAlpha, 51
- isAlphaNum, 51
- isAscii, 50
- isAsciiLower, 50
- isAsciiUpper, 51
- isBaseType, 214
- isBigComment, 271
- isBinDigit, 51
- isCase, 239, 264
- isCode, 271
- isComb, 238, 264
- isCombTypeConsCall, 237, 262
- isCombTypeConsPartCall, 237, 262
- isCombTypeFuncCall, 237, 262
- isCombTypeFuncPartCall, 237, 262
- isComment, 271
- isConsCall, 240, 265
- isConsId, 245
- isConsPartCall, 240, 265
- isConsPattern, 241, 266
- isControl, 51
- isDefined, 127, 139
- isDigit, 51
- isDrive, 65
- isEmpty, 107, 131, 143, 150, 154
- isEmptyFM, 146
- isEmptySetRBT, 156
- isEmptyTable, 158
- isEOF, 89
- isExternal, 235, 260
- isExtSeparator, 64
- isFree, 239, 264
- isFunctionCall, 240, 265
- isFunctionPartCall, 240, 265
- isFunctionalType, 214
- isFunctionType, 233, 258
- isGround, 240, 265
- isHexDigit, 51
- isInfixOf, 102
- isInfixOp, 245, 254
- isIOReturnType, 214

- isIOType, 214
- isJust, 104
- isLatin1, 50
- isLeft, 61
- isLet, 239, 264
- isLetter, 271
- isListId, 245, 254
- isLit, 238, 263
- isLower, 51
- isMeta, 271
- isModuleHead, 271
- isNothing, 104
- isOctDigit, 51
- isOr, 239, 264
- isPathSeparator, 64
- isPolyType, 214
- isPosix, 135
- isPrefixOf, 102
- isPrelude, 215
- isqrt, 86
- isRelative, 67
- isRight, 61
- isRuleExternal, 236, 261
- isSearchPathSeparator, 64
- isSmallComment, 271
- isSpace, 51
- isSuffixOf, 102
- isTCons, 233, 258
- isText, 271
- isTupleId, 245, 254
- isTVar, 233, 258
- isTypeSyn, 231, 256
- isUpper, 51
- isValid, 67
- isVar, 138, 238, 263
- isWindows, 135
- italic, 118, 170

- joinDrive, 65
- joinModuleIdentifiers, 58
- joinPath, 67
- JSBranch, 94
- jsConstTerm, 94
- JSExp, 92

- JSFDecl, 94
- JSSStat, 93

- Key, 95
- keyOrder, 146
- KeyPred, 95
- keysFM, 146
- kics2, 9
- kics2rc, 16

- lab, 150
- lab', 151
- labEdges, 152
- labNode', 151
- labNodes, 152
- labUEdges, 153
- labUNodes, 153
- langle, 116
- larrow, 117
- last, 102
- LayoutChoice, 219
- lbrace, 116
- lbracket, 116
- ldeclsOfRule, 215
- LEdge, 147
- lefts, 61
- leqChar, 157
- leqCharIgnoreCase, 157
- leqLexGerman, 157
- leqList, 157
- leqString, 157
- leqStringIgnoreCase, 157
- let, 17
- letBinds, 238, 263
- letBody, 238, 263
- line, 108
- linebreak, 108
- linesep, 107
- list, 114
- list2ac, 218
- list2CategorizedHtml, 163
- ListBoxScroll, 85
- listenOn, 105, 133
- listenOnFresh, 133

- listPattern, 218
- listSpaced, 114
- listToDefaultArray, 142
- listToDeq, 143
- listToErrorArray, 142
- listToFM, 144
- listToMaybe, 104
- listType, 216
- litem, 171
- Literal, 228
- literal, 237, 262
- LNode, 147
- local installation, 6
- log, 69
- logBase, 69
- logfile, 60
- lookup, 154
- lookupAFCSbst, 267
- lookupFileInPath, 63
- lookupFlatCurryFileInLoadPath, 229
- lookupFM, 146
- lookupModuleSourceInLoadPath, 59
- lookupRBT, 158
- lookupSubst, 161
- lookupWithDefaultFM, 146
- lparen, 115
- LPath, 148
- lpre, 151
- lpre', 152
- lsuc, 150
- lsuc', 152

- magenta, 119
- MailOption, 178
- main, 53
- mainWUI, 188, 196
- makeRelative, 67
- makeValid, 67
- mapAccumES, 62
- mapAccumL, 103
- mapAccumR, 103
- mapChildFamilies, 159
- mapChildFamiliesIO, 160
- mapChildren, 159
- mapChildrenIO, 160
- mapES, 62
- mapFamily, 159
- mapFamilyIO, 160
- mapFM, 145
- mapMaybe, 104
- mapMMaybe, 104
- mapT, 98
- mapT_, 98
- mapValues, 132
- markdown, 28
- MarkdownDoc, 179
- MarkdownElem, 179
- markdownEscapeChars, 181
- markdownText2CompleteHTML, 181
- markdownText2CompleteLaTeX, 181
- markdownText2HTML, 181
- markdownText2LaTeX, 181
- markdownText2LaTeXWithFormat, 181
- match, 150
- matchAny, 149
- matchHead, 143
- matchLast, 143
- matrix, 82
- max3, 87
- maxFM, 146
- maximum, 103
- maximumBy, 103
- maxlist, 87
- maxValue, 132
- maybeToList, 104
- maybeType, 216
- MContext, 148
- MenuItem, 79
- mergeSort, 157
- min3, 87
- minFM, 146
- minimum, 103
- minimumBy, 103
- minlist, 87
- minusFM, 145
- minValue, 132
- missingArgs, 237, 262
- missingCombArgs, 238, 263

- mkGraph, [149](#)
- mkUGraph, [149](#)
- MName, [206](#)
- modify, [62](#)
- modifyIORef, [92](#)
- modNameToPath, [59](#)
- modsOfType, [214](#)
- mplus, [104](#)
- multipleSelection, [173](#)

- nav, [170](#)
- nnbsp, [169](#)
- neighbors, [150](#)
- neighbors', [151](#)
- nest, [108](#)
- newDBEntry, [99](#)
- newDBKeyEntry, [99](#)
- newIORef, [92](#)
- newNodes, [152](#)
- newTreeLike, [154](#)
- nextBoolean, [154](#)
- nextInt, [153](#)
- nextIntRange, [153](#)
- nmap, [153](#)
- noChildren, [159](#)
- Node, [147](#)
- node', [151](#)
- nodeRange, [150](#)
- nodes, [152](#)
- noGuard, [217](#)
- noHandlerPage, [177](#)
- noindex, [30](#)
- noNodes, [150](#)
- normalise, [67](#)
- notEmpty, [131](#)
- nub, [100](#)
- nubBy, [100](#)

- odd, [87](#)
- olist, [171](#)
- on, [71](#)
- once, [7](#)
- onlyindex, [30](#)
- OpDecl, [223](#)

- openFile, [88](#)
- operation
 - external, [278](#)
- opFixity, [234](#), [259](#)
- opName, [234](#), [259](#)
- opPrecedence, [234](#), [259](#)
- opt, [201](#)
- OptDescr, [72](#)
- optimize, [13](#)
- Option, [248](#)
- option
 - in source file, [15](#)
- Options, [219](#), [242](#)
- options, [219](#)
- orC, [53](#)
- orExps, [238](#), [263](#)
- out, [151](#)
- out', [152](#)
- outdeg, [151](#)
- outdeg', [152](#)
- overlapWarn, [60](#)

- page, [169](#)
- pageBodyAttr, [169](#)
- pageCSS, [168](#)
- pageEnc, [168](#)
- pageLinkInfo, [169](#)
- pageMetaInfo, [168](#)
- PageParam, [166](#)
- par, [170](#)
- parallel, [12](#)
- parens, [115](#)
- parensIf, [115](#)
- parseHtmlString, [178](#)
- Parser, [106](#)
- parser, [14](#)
- ParserRep, [106](#)
- parseXmlString, [199](#)
- partition, [52](#), [101](#)
- partitionEithers, [61](#)
- password, [173](#)
- patArgs, [241](#), [266](#)
- patCons, [241](#), [266](#)
- patExpr, [241](#), [266](#)

- Path, 148
- path, 8, 12
- pathSeparator, 64
- pathSeparatorChar, 63
- pathSeparators, 64
- patLiteral, 241, 266
- Pattern, 228
- pattern
 - functional, 18
- pChar, 218
- permutations, 101
- permute, 52
- persistentSQLite, 98
- pi, 69
- plainCode, 271
- PlClause, 122
- PlGoal, 122
- plList, 123
- PlTerm, 123
- plusFM, 145
- plusFM.C, 145
- pNil, 218
- popupMessage, 84
- Pos, 160
- pow, 86
- ppAVarIndex, 253
- ppBranch, 245, 253
- ppCaseType, 245, 253
- ppCExpr, 221
- ppCFuncDecl, 221
- ppCFuncDeclWithoutSig, 221
- ppCFuncSignature, 221
- ppCLiteral, 221
- ppComb, 244, 253
- ppConsDecl, 244, 252
- ppConsDecls, 244, 252
- ppConsExports, 243, 251
- ppCOpDecl, 220
- ppCPattern, 221
- ppCStatement, 221
- ppCTypeDecl, 220
- ppCTypeExpr, 221
- ppCurryProg, 220
- ppDecl, 245, 253
- ppDecls, 244, 253
- ppExp, 244, 253
- ppExports, 220, 243, 251
- ppExpr, 244, 253
- ppFixity, 243, 252
- ppFunc, 221
- ppFuncDecl, 244, 253
- ppFuncDecls, 244, 252
- ppFuncExports, 243, 252
- ppHeader, 243, 251
- ppImport, 243, 252
- ppImports, 220, 243, 252
- ppInfixOp, 254
- ppInfixQOp, 245
- ppLiteral, 244, 253
- ppMName, 220
- ppName, 245
- ppOpDecl, 243, 252
- ppOpDecls, 243, 252
- ppPattern, 245, 253
- ppPrefixOp, 245, 254
- ppPrefixQOp, 245
- ppProg, 243, 251
- ppQFunc, 221
- ppQName, 245, 254
- ppQType, 221
- pPrint, 107
- ppRule, 244, 253
- ppTVarIndex, 244, 252
- ppType, 221
- ppTypeDecl, 243, 252
- ppTypeDecls, 243, 252
- ppTypeExp, 244, 252
- ppTypeExport, 243, 251
- ppTypeExpr, 244, 252
- ppVarIndex, 244, 253
- prdfs, 12
- pre, 150, 171, 211
- pre', 151
- preprocessor, 35
- pretty, 120
- prettyCurryProg, 220
- printAllValuesWith, 128
- printMemInfo, 122

printValues, 133
 printValuesWith, 128
 ProcessInfo, 120
 product, 102
 profile, 13
 profileSpace, 122
 profileSpaceNF, 122
 profileTime, 122
 profileTimeNF, 122
 profiling, 6, 13
 Prog, 222
 progFuncs, 230, 255
 progImports, 230, 255
 progName, 230, 255
 progOps, 230, 255
 program
 documentation, 28
 testing, 33
 progTypes, 230, 255
 prompt, 13
 ProtocolMsg, 49
 publicConsNames, 213
 publicFuncNames, 213
 publicTypeNames, 213
 punctuate, 112
 pureio, 13
 puts, 62
 pVars, 218

 QName, 206, 222
 Qualification, 219
 QualMode, 242
 qualMode, 242
 Query, 95
 Queue, 142
 quickSort, 157
 quiet, 60

 radio_main, 173
 radio_main_off, 173
 radio_other, 173
 range, 233, 258
 rangle, 116
 rarrow, 117

 rbrace, 116
 rbracket, 116
 rcFileContents, 58
 rcFileName, 58
 rcParams, 59
 readAbstractCurryFile, 212
 readCgiServerMsg, 177
 readCompleteFile, 91
 readCSV, 54
 readCSVFile, 54
 readCSVFileWithDelims, 54
 readCSVWithDelims, 54
 readCurry, 47, 212
 readCurryWithImports, 212
 readCurryWithParseOptions, 212
 readFileWithXmlDocs, 198
 readFlatCurry, 47, 228
 readFlatCurryFile, 229
 readFlatCurryInt, 229
 readFlatCurryIntWithImports, 246
 readFlatCurryIntWithImportsInPath, 246
 readFlatCurryWithImports, 246
 readFlatCurryWithImportsInPath, 246
 readFlatCurryWithParseOptions, 228
 readFM, 147
 readGlobal, 73
 readHex, 124, 125
 readHtmlFile, 178
 readInt, 124
 readIORef, 92
 readNat, 124
 readOct, 125
 readPropertyFile, 124
 readQTerm, 126
 readQTermFile, 126
 readQTermListFile, 126
 readScan, 271
 readsQTerm, 126
 readsTerm, 126
 readsUnqualifiedTerm, 125
 readTerm, 126
 readUnqualifiedTerm, 126
 readUnsafeXmlFile, 198
 readUntypedCurry, 212

[readUntypedCurryWithParseOptions](#), 212
[readXmlFile](#), 198
[recip](#), 69
[ReconfigureItem](#), 77
[red](#), 119
[RedBlackTree](#), 154
[redirect](#), 168
[registerCgiServer](#), 177
[registerPort](#), 53
[removeDirectory](#), 56
[removeEscapes](#), 180
[removeFile](#), 56
[removeRegionStyle](#), 84
[renameDirectory](#), 56
[renameFile](#), 56
[Rendering](#), 182, 189
[renderList](#), 188, 196
[renderTaggedTuple](#), 188, 196
[renderTuple](#), 188, 196
[rep](#), 201
[replace](#), 102
[replaceBaseName](#), 66
[replaceChildren](#), 159
[replaceChildrenIO](#), 160
[replaceDirectory](#), 66
[replaceExtension](#), 65
[replaceFileName](#), 66
[repSeq1](#), 202
[repSeq2](#), 203
[repSeq3](#), 203
[repSeq4](#), 204
[repSeq5](#), 204
[repSeq6](#), 205
[RequiredSpec](#), 249
[requires](#), 249
[resetbutton](#), 172
[resultType](#), 214, 234, 259
[returnES](#), 62
[returnT](#), 97
[rights](#), 61
[rnmAllVars](#), 240, 265
[rnmAllVarsInFunc](#), 236, 261
[rnmAllVarsInProg](#), 231, 256
[rnmAllVarsInRule](#), 237, 262
[rnmAllVarsInTypeExpr](#), 234, 259
[rnmProg](#), 231, 256
[rotate](#), 143
[round](#), 69
[row](#), 82
[rparen](#), 115
[rts](#), 15
[Rule](#), 224
[ruleArgs](#), 236, 261
[ruleBody](#), 236, 261
[ruleExtDecl](#), 236, 261
[runCgiServerCmd](#), 177
[runConfigControlledGUI](#), 82
[runControlledGUI](#), 82
[runcurry](#), 40
[runFormServerWithKey](#), 175
[runFormServerWithKeyAndFormParams](#), 175
[runGUI](#), 82
[runGUIwithParams](#), 82
[runHandlesControlledGUI](#), 83
[runInitControlledGUI](#), 83
[runInitGUI](#), 82
[runInitGUIwithParams](#), 82
[runInitHandlesControlledGUI](#), 83
[runJustT](#), 97
[runPassiveGUI](#), 82
[runQ](#), 96
[runT](#), 96

[safe](#), 14
[satisfy](#), 107
[scan](#), 271
[scanl](#), 103
[scanl1](#), 103
[scanr](#), 103
[scanr1](#), 103
[scc](#), 155
[sClose](#), 105, 134
[searchPathSeparator](#), 64
[SearchTree](#), 7, 127, 138
[searchTreeSize](#), 127, 139
[second](#), 71
[section](#), 170
[SeekMode](#), 88

seeText, 84
 select, 132
 selection, 173
 selectionInitial, 173
 selectValue, 132
 semi, 116
 semiBraces, 114
 semiBracesSpaced, 114
 sendMail, 179
 sendMailWithOptions, 179
 sep, 112
 separatorChar, 63
 seq1, 202
 seq2, 203
 seq3, 203
 seq4, 204
 seq5, 204
 seq6, 205
 seqStrActions, 50
 sequenceMaybe, 104
 sequenceT, 97
 sequenceT_, 98
 set, 114
 set functions, 7
 set0, 130
 set0With, 130
 set1, 130
 set1With, 130
 set2, 130
 set2With, 130
 set3, 130
 set3With, 130
 set4, 130
 set4With, 130
 set5, 131
 set5With, 131
 set6, 131
 set6With, 131
 set7, 131
 set7With, 131
 setAssoc, 91
 setConfig, 83
 setCurrentDirectory, 55
 setEnviron, 134
 setExtended, 60
 setFullPath, 60
 setHtmlDir, 60
 setIndentWith, 220
 setInsertEquivalence, 155
 setLayoutChoice, 220
 setLogfile, 60
 setModName, 220
 setOverlapWarn, 60
 setPageWith, 219
 setQualification, 220
 setQuiet, 59
 SetRBT, 155
 setRBT2list, 156
 setSpaced, 114
 setSpecials, 60
 setValue, 83
 showAFCSbst, 267
 showCProg, 220
 showCSV, 54
 showCurryExpr, 247
 showCurryId, 247
 showCurryType, 247
 showCurryVar, 247
 showEscape, 253
 showFlatFunc, 247
 showFlatProg, 246
 showFlatType, 246
 showFM, 147
 showGraph, 153
 showHtmlExp, 174
 showHtmlExps, 174
 showHtmlPage, 174
 showJSExp, 94
 showJSFDecl, 94
 showJSStat, 94
 showLatexDoc, 175
 showLatexDocs, 176
 showLatexDocsWithPackages, 176
 showLatexDocWithPackages, 175
 showLatexExp, 175
 showLatexExps, 175
 showMemInfo, 122
 showPlClause, 123

[showPlGoal](#), 123
[showPlGoals](#), 123
[showPlProg](#), 123
[showPlTerm](#), 123
[showQNameInModule](#), 228
[showQTerm](#), 125
[showSearchTree](#), 127, 139
[showSubst](#), 161
[showTerm](#), 125
[showTError](#), 99
[showTestCase](#), 50
[showTestCompileError](#), 50
[showTestEnd](#), 50
[showTestMod](#), 50
[showXmlDoc](#), 198
[showXmlDocWithParams](#), 198
[simpleRule](#), 217
[sin](#), 70
[singleton variables](#), 7
[sinh](#), 70
[sizedSubset](#), 52
[sizeFM](#), 145
[sleep](#), 135
[snoc](#), 143
[Socket](#), 105, 133
[socketAccept](#), 105, 133
[socketName](#), 105
[softbreak](#), 108
[softline](#), 108
[solveAll](#), 7
[some](#), 107
[someDBInfos](#), 98
[someDBKeyInfos](#), 99
[someDBKeyProjections](#), 99
[someDBKeys](#), 98
[someSearchTree](#), 127, 139
[someSolution](#), 68
[someValue](#), 68, 129, 140
[someValueWith](#), 129, 140
[sort](#), 155
[sortBy](#), 102
[sortRBT](#), 156
[sortValues](#), 133
[sortValuesBy](#), 133
[source-file option](#), 15
[space](#), 116
[specials](#), 60
[spiceup](#), 43
[Spicey](#), 43
[split](#), 101
[splitBaseName](#), 63
[splitDirectories](#), 67
[splitDirectoryBaseName](#), 63
[splitDrive](#), 65
[splitExtension](#), 64
[splitExtensions](#), 65
[splitFileName](#), 66
[splitFM](#), 145
[splitModuleFileName](#), 58
[splitModuleIdentifiers](#), 58
[splitOn](#), 101
[splitPath](#), 63, 66
[splitSearchPath](#), 64
[splitSet](#), 52
[sqrt](#), 69
[squote](#), 116
[squotes](#), 114
[standardForm](#), 168
[standardPage](#), 169
[star](#), 107
[stderr](#), 88
[stdin](#), 88
[stdout](#), 88
[Strategy](#), 127, 138
[string](#), 115, 200
[string2ac](#), 218
[string2urlencoded](#), 175
[stringList2ItemList](#), 163
[stringPattern](#), 218
[stringType](#), 216
[stripCurrySuffix](#), 58
[stripSuffix](#), 63
[strong](#), 170
[Style](#), 80
[style](#), 172
[styleSheet](#), 172
[submitForm](#), 177
[subset](#), 52

- Subst, 161
- subst, 268
- substBranch, 268
- substExpr, 268
- substFunc, 268
- substPattern, 268
- substRule, 268
- substSnd, 268
- suc, 150
- suc', 151
- suffixSeparatorChar, 63
- sum, 102
- supply, 12
- system, 135

- table, 171
- TableRBT, 158
- tableRBT2list, 158
- tabulator stops, 7
- tagOf, 198
- tails, 102
- takeBaseName, 66
- takeDirectory, 66
- takeDrive, 65
- takeExtension, 64
- takeExtensions, 65
- takeFileName, 66
- tan, 70
- tanh, 70
- tConsArgs, 233, 258
- tConsName, 233, 258
- teletype, 171
- Term, 161
- TermEq, 160
- TermEqs, 160
- terminal, 106
- TError, 96
- TErrorKind, 96
- testing programs, 33
- testScan, 271
- text, 107
- textarea, 173
- TextEditScroll, 85
- textfield, 172

- textOf, 198
- textOfXml, 198
- textstyle, 172
- tilde, 117
- time, 13
- toCalendarTime, 136
- toClockTime, 136
- toDayString, 137
- Token, 270
- Tokens, 270
- toLower, 51
- toTimeString, 137
- toUpper, 51
- toUTCTime, 136
- toVar, 218
- trace, 13, 54, 138
- traceId, 54
- traceIO, 55
- traceShow, 54
- traceShowId, 55
- Transaction, 95
- transformQ, 96
- transformWSpec, 183, 190
- transpose, 101
- Traversable, 158
- trBranch, 240, 265
- trCombType, 237, 262
- trCons, 232, 257
- tree2list, 155
- trExpr, 239, 264
- trFunc, 235, 260
- trOp, 234, 259
- trPattern, 241, 266
- trProg, 230, 255
- trRule, 236, 261
- trType, 231, 256
- trTypeExpr, 233, 258
- truncate, 69
- try, 7
- tryParse, 212
- tryReadACYFile, 213
- tryReadCurryFile, 212
- tryReadCurryWithImports, 212
- tupled, 114

- tupledSpaced, 114
- tupleExpr, 217
- tuplePattern, 218
- tupleType, 216
- TVarIndex, 222
- tVarIndex, 233, 258
- tvarsOfType, 214
- typeCons, 214
- typeConsDecls, 231, 256
- TypeDecl, 222
- TypeEnv, 269
- TypeExpr, 223
- typeName, 214, 231, 256
- typeParams, 231, 256
- types, 213
- typeSyn, 231, 256
- typeVis, 214
- typeVisibility, 231, 256

- UContext, 148
- UDecomp, 148
- UEdge, 147
- unfold, 152
- UGr, 148
- ulist, 171
- unAnnExpr, 267
- unAnnFuncDecl, 267
- unAnnPattern, 267
- unAnnProg, 267
- unAnnRule, 267
- underline, 119
- unfoldr, 103
- UnificationError, 162
- unify, 162
- union, 100
- unionBy, 100
- unionRBT, 156
- unitFM, 144
- unitType, 216
- UNode, 147
- unregisterCgiServer, 177
- unregisterPort, 53
- unsafePerformIO, 138
- unscan, 271

- unsetEnviron, 134
- untypedAbstractCurryFileName, 212
- UPath, 148
- Update, 229, 254
- update, 142, 154
- updateDBEntry, 99
- updateFile, 91
- updatePropertyFile, 124
- updateRBT, 158
- updateValue, 83
- updateXmlFile, 199
- updBranch, 240, 266
- updBranches, 239, 265
- updBranchExpr, 241, 266
- updBranchPattern, 240, 266
- updCases, 239, 265
- updCombs, 239, 264
- updCons, 232, 257
- updConsArgs, 233, 258
- updConsAriety, 232, 257
- updConsName, 232, 257
- updConsVisibility, 233, 258
- updFM, 145
- updFrees, 239, 264
- updFunc, 235, 260
- updFuncArgs, 236, 261
- updFuncAriety, 235, 260
- updFuncBody, 236, 261
- updFuncName, 235, 260
- updFuncRule, 235, 260
- updFuncType, 235, 260
- updFuncTypes, 234, 259
- updFuncVisibility, 235, 260
- updLets, 239, 264
- updLiterals, 239, 264
- updOp, 234, 259
- updOpFixity, 234, 259
- updOpName, 234, 259
- updOpPrecedence, 234, 259
- updOrs, 239, 264
- updPatArgs, 241, 266
- updPatCons, 241, 266
- updPatLiteral, 241, 266
- updPattern, 241, 266

- updProg, [230](#), [255](#)
- updProgExps, [231](#), [256](#)
- updProgFuncs, [230](#), [255](#)
- updProgImports, [230](#), [255](#)
- updProgName, [230](#), [255](#)
- updProgOps, [230](#), [255](#)
- updProgTypes, [230](#), [255](#)
- updQNames, [240](#), [265](#)
- updQNamesInConsDecl, [233](#), [258](#)
- updQNamesInFunc, [236](#), [261](#)
- updQNamesInProg, [231](#), [256](#)
- updQNamesInRule, [237](#), [262](#)
- updQNamesInType, [232](#), [257](#)
- updQNamesInTypeExpr, [234](#), [259](#)
- updRule, [236](#), [261](#)
- updRuleArgs, [236](#), [262](#)
- updRuleBody, [237](#), [262](#)
- updRuleExtDecl, [237](#), [262](#)
- updTCons, [233](#), [258](#)
- updTVars, [233](#), [258](#)
- updType, [231](#), [256](#)
- updTypeConsDecls, [232](#), [257](#)
- updTypeds, [240](#), [265](#)
- updTypeName, [231](#), [256](#)
- updTypeParams, [232](#), [257](#)
- updTypeSynonym, [232](#), [257](#)
- updTypeVisibility, [232](#), [257](#)
- updVars, [239](#), [264](#)
- urlencoded2string, [174](#)
- usageInfo, [72](#)

- v, [13](#)
- validDate, [137](#)
- valueOf, [131](#)
- Values, [130](#)
- values2list, [133](#)
- ValueSequence, [141](#)
- variables
 - singleton, [7](#)
- varId, [139](#)
- VarIdx, [160](#)
- VarIndex, [222](#)
- varNr, [237](#), [262](#)
- varsOfExp, [215](#)
- varsOfFDecl, [215](#)
- varsOfLDecl, [215](#)
- varsOfPat, [215](#)
- varsOfRhs, [215](#)
- varsOfRule, [215](#)
- varsOfStat, [215](#)
- vcat, [112](#)
- verbatim, [171](#)
- verbosity, [13](#)
- version, [211](#)
- Visibility, [222](#)
- vsep, [111](#)
- vsepBlank, [111](#)
- vsToList, [141](#)

- w10Tuple, [186](#), [194](#)
- w11Tuple, [186](#), [194](#)
- w12Tuple, [186](#), [195](#)
- w4Tuple, [185](#), [192](#)
- w5Tuple, [185](#), [192](#)
- w6Tuple, [185](#), [193](#)
- w7Tuple, [185](#), [193](#)
- w8Tuple, [185](#), [193](#)
- w9Tuple, [186](#), [194](#)
- waitForSocketAccept, [105](#), [134](#)
- wCheckBool, [184](#), [191](#)
- wCheckMaybe, [187](#), [196](#)
- wCons10, [186](#), [194](#)
- wCons10JS, [194](#)
- wCons11, [186](#), [194](#)
- wCons11JS, [195](#)
- wCons12, [187](#), [195](#)
- wCons12JS, [195](#)
- wCons2, [184](#), [191](#)
- wCons2JS, [192](#)
- wCons3, [185](#), [192](#)
- wCons3JS, [192](#)
- wCons4, [185](#), [192](#)
- wCons4JS, [192](#)
- wCons5, [185](#), [192](#)
- wCons5JS, [192](#)
- wCons6, [185](#), [193](#)
- wCons6JS, [193](#)
- wCons7, [185](#), [193](#)

wCons7JS, 193
wCons8, 186, 193
wCons8JS, 193
wCons9, 186, 194
wCons9JS, 194
wConstant, 183, 190
wEither, 187, 196
where, 17
wHidden, 183, 190
white, 119
wHList, 187, 195
Widget, 74
WidgetRef, 80
wInt, 183, 190
withCondition, 183, 189
withConditionJS, 190
withConditionJSName, 190
withError, 183, 189
withRendering, 182, 189
wJoinTuple, 187, 191
wList, 187, 195
wListWithHeadings, 187, 195
wMatrix, 187, 195
wMaybe, 187, 195
wMultiCheckSelect, 184, 191
wPair, 184, 191
wRadioBool, 184, 191
wRadioMaybe, 187, 196
wRadioSelect, 184, 191
wRequiredString, 183, 190
wRequiredStringSize, 183, 190
writeAbstractCurryFile, 213
writeAssertResult, 50
writeCSVFile, 54
writeFCY, 229
writeGlobal, 73
writeIORef, 92
writeQTermFile, 126
writeQTermListFile, 126
writeXmlFile, 198
writeXmlFileWithParams, 198
wSelect, 184, 191
wSelectBool, 184, 191
wSelectInt, 184, 191
wString, 183, 190
wStringSize, 183, 190
wTextArea, 183, 190
WTree, 182, 189
wTree, 188, 196
wTriple, 184, 192
wui2html, 188, 196
WuiHandler, 182, 189
wuiHandler2button, 182, 189
wuiInForm, 188, 196
WuiSpec, 182, 189
wuiWithErrorForm, 188, 196
XAttrConv, 199
XElemConv, 199
xml, 198
xml2FlatCurry, 247
XmlDocParams, 197
XmlExp, 197
xmlFile2FlatCurry, 247
xmlRead, 200
XmlReads, 199
xmlReads, 200
xmlShow, 200
XmlShows, 199
xmlShows, 200
XOptConv, 199
XPrimConv, 199
XRepConv, 199
txtxt, 198
yellow, 119