Pure Language and Library Documentation

Release 0.56

Albert Gräf (Editor)

September 26, 2012

Contents

This manual collects all of Pure's online documentation: *The Pure Manual* which covers the Pure language and the operation of the Pure interpreter; the *Pure Library Manual* which describes the standard library modules included in the distribution of the Pure interpreter; all available documentation for the various addon modules which can be downloaded as separate packages from the Pure website; and an appendix with *installation instructions* and additional information for *Windows users*.

Most of the Pure documentation is distributed under the GNU Free Documentation License. The authors of the current edition are listed below. (This just lists the primary section authors in alphabetical order; please check the different parts of this manual for additional authorship and licensing information.)

- Albert Gräf (*The Pure Manual*; *Pure Library Manual*; various addon manuals)
- Rob Hubbard (*Pure-Rational - Rational number library for the Pure programming language*)
- Kay-Uwe Kirstein (*Gnuplot bindings*)
- Eddie Rucker (*Pure-CSV - Comma Separated Value Interface for the Pure Programming Language*; *pure-gsl - GNU Scientific Library Interface for Pure*)
- Jiri Spitz (*Pure-GLPK - GLPK interface for the Pure programming language*)
- Peter Summerland (*Pure-Sql3*, *pure-stlmap*, *pure-stlvec*)

The Pure programming system is free and open source software. The interpreter runtime, the standard library and most of the addon modules are distributed under the GNU Lesser General Public License or the 3-clause BSD License which allow for commercial applications. Some parts of the system also use the GNU General Public License (typically because they interface to other GPL'ed software such as Gnumeric, GSL and Octave). Details about authorship and license conditions can be found in the sources or in the various manual sections.

For more information, discussions, feedback, questions, suggestions etc. please see:

- Pure website: http://pure-lang.googlecode.com
- Pure mailing list: http://groups.google.com/group/pure-lang

Chapter 1

The Pure Manual

Version 0.56, September 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

Copyright (c) 2009-2012 by Albert Gräf. This document is available under the GNU Free Documentation License. Also see the Copying section for licensing information of the software.

This manual describes the Pure programming language and how to invoke the Pure interpreter program. To read the manual inside the interpreter, just type `help` at the command prompt. See the Online Help section for details.

There is a companion to this manual, the *Pure Library Manual* which contains the description of the standard library operations. More information about Pure and the latest sources can be found under the following URLs:

- Pure website: http://pure-lang.googlecode.com
- Pure mailing list: http://groups.google.com/group/pure-lang

Information about how to install Pure can be found in the document *Installing Pure (and LLVM)*.

1.1 Introduction

Pure is a functional programming language based on term rewriting. This means that all your programs are essentially just collections of symbolic equations which the interpreter uses to reduce expressions to their simplest ("normal") form. This makes for a rather powerful and flexible programming model featuring dynamic typing and general polymorphism. In addition, Pure programs are compiled to efficient native code on the fly, using the LLVM compiler framework, so programs are executed reasonably fast and interfacing to C is very easy. If you have the necessary 3rd party compilers installed then you can even inline functions written in C and a number of other languages and call them just like any other Pure

function. The ease with which you can interface to 3rd party software makes Pure useful for a wide range of applications from symbolic algebra and scientific programming to database, web and multimedia applications.

The Pure language is implemented by the **Pure interpreter** program. Just like other programming language interpreters, the Pure interpreter provides an interactive environment in which you can type definitions and expressions, which are executed as you type them at the interpreter's command prompt. However, despite its name the Pure interpreter never really "interprets" any Pure code. Rather, it acts as a frontend to the **Pure compiler**, which takes care of incrementally compiling Pure code to native (machine) code. This has the benefit that the compiled code runs much faster than the usual kinds of "bytecode" that you find in traditional programming language interpreters.

You can use the interpreter as a sophisticated kind of "desktop calculator" program. Simply run the program from the shell as follows:

```
$ pure

 __ \  |   |  __| _ \    Pure 0.56 (x86_64-unknown-linux-gnu)
 |   | |   | |    __/    Copyright (c) 2008-2012 by Albert Graef
 .__/ \__,_|_|  \___|    (Type 'help' for help, 'help copying'
_|                       for license information.)

Loaded prelude from /usr/lib/pure/prelude.pure.

>
```

The interpreter prints its sign-on message and leaves you at its "> " command prompt, where you can start typing definitions and expressions to be evaluated:

```
> 17/12+23;
24.4166666666667
> fact n = if n>0 then n*fact (n-1) else 1;
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Typing the `quit` command or the end-of-file character (`Ctrl-d` on Unix systems) at the beginning of the command line exits the interpreter and takes you back to the shell.

Instead of typing definitions and evaluating expressions in an interactive fashion as shown above, you can also put the same code in an (ASCII or UTF-8) text file called a **Pure program** or **script** which can then be executed by the interpreter in "batch mode", or compiled to a standalone executable which can be run directly from the command line. As an aid for writing script files, a bunch of syntax highlighting files and programming modes for various popular text editors are included in the Pure sources.

More information about invoking the Pure interpreter can be found in the Invoking Pure section below. This is followed by a description of the Pure language in Pure Overview and subsequent sections, including an extensive Examples section which can serve as a mini-tutorial on Pure. The interactive facilities of the Pure interpreter are discussed in the Interactive Usage section, while the Batch Compilation section explains how to translate Pure programs to native executables and a number of other object file formats. The Caveats and

Notes section discusses useful tips and tricks, as well as various pitfalls and how to avoid them. The manual concludes with some authorship and licensing information and pointers to related software.

1.1.1 Further Reading

This manual is not intended as a general introduction to functional programming, so at least some familiarity with this programming style is assumed. If Pure is your first functional language then you might want to look at the Functional Programming wikipedia article to see what it is all about and find pointers to current literature on the subject. In any case we hope that you'll find Pure helpful in exploring functional programming, as it is fairly easy to learn but a very powerful language.

As already mentioned, Pure uses term rewriting as its underlying computational model, which goes well beyond functional programming in some ways. Term rewriting has long been used in computer algebra systems, and Michael O'Donnell pioneered its use as a programming language already in the 1980s. But until recently implementations have not really been efficient enough to be useful as general-purpose programming languages; Pure strives to change that. A good introduction to the theory of the term rewriting calculus and its applications is the book by Baader and Nipkow.

1.1.2 Typographical Conventions

Program examples are always set in typewriter font. Here's how a typical code sample may look like:

```
fact n = if n>0 then n*fact(n-1) else 1;
```

These can either be saved to a file and then loaded into the interpreter, or you can also just type them directly in the interpreter. If some lines start with the interpreter prompt "> ", this indicates an example interaction with the interpreter. Everything following the prompt (excluding the "> " itself) is meant to be typed exactly as written. Lines lacking the "> " prefix show results printed by the interpreter. Example:

```
> fact n = if n>0 then n*fact(n-1) else 1;
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Similarly, lines starting with the "$ " prompt indicate shell interactions. For instance,

```
$ pure
```

indicates that you should type the command `pure` on your system's command line.

The grammar notation in this manual uses an extended form of BNF (Backus-Naur form), which looks as follows:

```
expression  ::=   "{" expr_list (";" expr_list)* [";"] "}"
expr_list   ::=   expression (',' expression)*
```

Parentheses are used to group syntactical elements, while brackets denote optional elements. We also use the regular expression operators * and + to denote repetitions (as usual, * denotes zero or more, + one or more repetitions of the preceding element). Terminals (literal elements such as keywords and delimiters) are enclosed in double or single quotes.

These EBNF rules are used for both lexical and syntactical elements, but note that the former are concerned with entities formed from single characters and thus tokens are meant to be typed exactly as written, whereas the latter deal with larger syntactical structures where whitespace between tokens is generally insignificant.

1.2 Invoking Pure

The Pure interpreter is invoked as follows:

```
pure [options ...] [script ...] [-- args ...]
pure [options ...] -x script [args ...]
```

Use `pure -h` to get help about the command line options. As already mentioned, just the `pure` command without any command line parameters invokes the interpreter in interactive mode, see Running Interactively below for details. Some other important ways to invoke the interpreter are summarized below.

`pure -g` Runs the interpreter interactively, with debugging support.

`pure script ...` Runs the given scripts in batch mode.

`pure -i script ...` Runs the given scripts in batch mode as above, but then enters the interactive command loop. (Add `-g` to also get debugging support, and `-q` to suppress the sign-on message.)

`pure -x script [arg ...]` Runs the given script with the given parameters. The script name and command line arguments are available in the global `argv` variable.

`pure -c script [-o prog]` Batch compilation: Runs the given script, compiling it to a native executable `prog` (a.out by default).

Depending on your local setup, there may be additional ways to run the Pure interpreter. In particular, if you have Emacs Pure mode installed, then you can just open a script in Emacs and run it with the `C-c C-k` keyboard command. For Emacs aficionados, this is probably the most convenient way to execute a Pure script interactively in the interpreter. Pure mode actually turns Emacs into an advanced IDE (integrated development environment) for Pure, which offers a lot of convenient features such as syntax highlighting, automatic indentation, online help and different ways to interact with the Pure interpreter.

1.2.1 Options

The interpreter accepts various options which are described in more detail below.

-c
Batch compilation.

--ctags
--etags
Create a tags file in ctags (vi) or etags (emacs) format.

--disable optname
Disable source option (conditional compilation).

--eager-jit
Enable eager JIT compilation. This requires LLVM 2.7 or later, otherwise this flag will be ignored.

--enable optname
Enable source option (conditional compilation).

--escape char
Interactive commands are prefixed with the specified character. Permitted prefixes are: `!$%&*,:<>@\|`.

-fPIC
-fpic
Create position-independent code (batch compilation).

-g
Enable symbolic debugging.

-h
--help
Print help message and exit.

-i
Force interactive mode (read commands from stdin).

-I directory
Add a directory to be searched for included source scripts.

-L directory
Add a directory to be searched for dynamic libraries.

-l libname
Library to be linked in batch compilation.

--main name
Name of main entry point in batch compilation.

--noediting
Disable command-line editing.

-n
--noprelude
Do not load the prelude.

--norc
Do not run the interactive startup files.

-o `filename`
Output filename for batch compilation.

-q
Quiet startup (suppresses sign-on message in interactive mode).

-T `filename`
Tags file to be written by `--ctags` or `--etags`.

-u
Do not strip unused functions in batch compilation.

-v`[level]`
Set verbosity level.

--version
Print version information and exit.

-w
Enable compiler warnings.

-x
Execute script with given command line arguments.

--
Stop option processing and pass the remaining command line arguments in the `argv` variable.

Besides these, the interpreter also understands a number of other command line switches for setting various compilation options; please see Compilation Options below for details.

Note: Option parsing follows the usual (Unix) conventions, but is somewhat more rigid than the GNU getopt conventions. In particular, it is *not* possible to combine short options, and there are no abbreviations for "long" options. Mixing options and other command line parameters is generally possible, but note that all option processing stops right after `-x` and `--`, passing the remaining parameters to the executing script in the Pure `argv` variable.

As usual, if an option takes a required argument, the argument may be written either as a separate command line parameter immediately following the option (as in `-I` *directory* or `--enable` *optname*), or directly after the option (`-I`*directory* or `--enable=`*optname*; note the equals sign in the case of a long option). Options with optional arguments work in the same fashion, but in this case the argument, if present, *must* be written directly behind the option.

1.2.2 Overview of Operation

If any source scripts are specified on the command line, they are loaded and executed, after which the interpreter exits. Otherwise the interpreter enters the interactive read-eval-print loop, see Running Interactively below. You can also use the `-i` option to enter the interactive loop (continue reading from stdin) even after processing some source scripts.

Options and source files are processed in the order in which they are given on the command line. Processing of options and source files ends when either the `--` or the `-x` option is encountered. The `-x` option must be followed by the name of a script to be executed, which becomes the "main script" of the application. In either case, any remaining parameters are passed to the executing script by means of the global `argc` and `argv` variables, denoting the number of arguments and the list of the actual parameter strings, respectively. In the case of `-x` this also includes the script name as `argv!0`. The `-x` option is useful, in particular, to turn Pure scripts into executable programs by including a "shebang" like the following as the first line in your main script. (This trick only works with Unix shells, though.)

```
#!/usr/local/bin/pure -x
```

The following variables are always predefined by the interpreter:

variable `argc`
variable `argv`
: The number of extra command line arguments and the arguments themselves as a list of strings; see above. These are useful if a script is usually run non-interactively and takes its input from the command line.

variable `compiling`
: A flag indicating whether the program is executed in a batch compilation (`-c` option), see Compiling Scripts below.

variable `version`
variable `sysinfo`
: The version string of the Pure interpreter and a string identifying the host system. These are useful if parts of your script depend on the particular version of the interpreter and the system it runs on. (An alternative way to deal with version and system dependencies is to use conditional compilation; see Conditional Compilation.)

If available, the prelude script prelude.pure is loaded by the interpreter prior to any other definitions, unless the `-n` or `--noprelude` option is specified. The prelude is searched for in the directory specified with the `PURELIB` environment variable. If the `PURELIB` variable is not set, a system-specific default is used. Relative pathnames of other source scripts specified on the command line are interpreted relative to the current working directory. In addition, the executed program may load other scripts and libraries via a `using` declaration in the source, which are searched for in a number of locations, including the directories named with the `-I` and `-L` options; see the Declarations and C Interface sections for details.

1.2.3 Compiling Scripts

The interpreter compiles scripts, as well as definitions that you enter interactively, automatically. This is done in an incremental fashion, as the code is needed, and is therefore known as JIT (**just in time**) compilation. Thus the interpreter never really "interprets" the source program or some intermediate representation, it just acts as a frontend to the compiler, taking care of compiling source code to native machine code before it gets executed.

Pure's LLVM backend does "lazy JIT compilation" by default, meaning that each function (global or local) is compiled no sooner than it is run for the first time. With the `--eager-jit` option, however, it will also compile all other (global or local) functions that may be called by the compiled function. (The `PURE_EAGER_JIT` environment variable, when set to any value, has the same effect, so that you do not have to specify the `--eager-jit` option each time you run the interpreter.) Eager JIT compilation may be more efficient in some cases (since bigger chunks of compilation work can be done in one go) and less efficient in others (e.g., eager JITing may compile large chunks of code which aren't actually called later, except in rare circumstances).

Note that the eager JIT mode is only available with LLVM 2.7 or later; otherwise this option will be ignored.

It is also possible to compile your scripts to native code beforehand, using the `-c` batch compilation option. This options forces the interpreter to non-interactive mode (unless `-i` is specified as well, which overrides `-c`). Any scripts specified on the command line are then executed as usual, but after execution the interpreter takes a snapshot of the program and compiles it to one of several supported output formats, LLVM assembler (.ll) or bitcode (.bc), native assembler (.s) or object (.o), or a native executable, depending on the output filename specified with `-o`. If the output filename ends in the .ll extension, an LLVM assembler file is created which can then be processed with the LLVM toolchain. If the output filename is just '-', the assembler file is written to standard output, which is useful if you want to pass the generated code to the LLVM tools in a pipeline. If the output filename ends in the .bc extension, an LLVM bitcode file is created instead.

The .ll and .bc formats are supported natively by the Pure interpreter, no external tools are required to generate these. If the target is an .s, .o or executable file, the Pure interpreter creates a temporary bitcode file on which it invokes the LLVM tools **opt** and **llc** to create a native assembler file, and then uses the C/C++ compiler (normally **gcc**, but you can change this with the `CC` and `CXX` environment variables) to assemble and link the resulting program (if requested). You can also specify additional libraries to be linked into the executable with the `-l` option. If the output filename is omitted, it defaults to a.out (a.exe on Windows).

The `-c` option provides a convenient way to quickly turn a Pure script into a standalone executable which can be invoked directly from the shell. One advantage of compiling your script is that this eliminates the JIT compilation time and thus considerably reduces the startup time of the program. Another reason to prefer a standalone executable is that it lets you deploy the program on systems without a full Pure installation (usually only the runtime library is required on the target system). On the other hand, compiled scripts also have some limitations, mostly concerning the use of the built-in `eval` function. Please see the Batch Compilation section for details.

The `-v64` (or `-v0100`) verbosity option can be used to have the interpreter print the commands it executes during compilation, see Verbosity and Debugging Options below. When creating an object file, this also prints the suggested linker command (including all the dynamic modules loaded by the script, which also have to be linked in to create a working executable), to which you only have to add the options describing the desired output file.

1.2.4 Tagging Scripts

Pure programs often have declarations and definitions of global symbols scattered out over many different source files. The *`--ctags`* and *`--etags`* options let you create a tags file which allows you to quickly locate these items in text editors such as **vi** and **emacs** which support this feature.

If *`--ctags`* or *`--etags`* is specified, the interpreter enters a special mode in which it only parses source files without executing them and collects information about the locations of global symbol declarations and definitions. The collected information is then written to a tags file in the ctags or etags format used by **vi** and **emacs**, respectively. The desired name of the tags file can be specified with the *`-T`* option; it defaults to tags for *`--ctags`* and TAGS for *`--etags`* (which matches the default tags file names used by **vi** and **emacs**, respectively).

The tags file contains information about the global constant, variable, macro, function and operator symbols of all scripts specified on the command line, as well as the prelude and other scripts included via a `using` clause. Tagged scripts which are located in the same directory as the tags file (or, recursively, in one of its subdirectories) are specified using relative pathnames, while scripts outside this hierarchy (such as included scripts from the standard library) are denoted with absolute pathnames. This scheme makes it possible to move an entire directory together with its tags file and have the tags information still work in the new location.

1.2.5 Running Interactively

If the interpreter runs in interactive mode, it repeatedly prompts you for input (which may be any legal Pure code or some special interpreter commands provided for interactive usage), and prints computed results. This is also known as the **read-eval-print** loop and is described in much more detail in the Interactive Usage section. To exit the interpreter, just type the `quit` command or the end-of-file character (`Ctrl-d` on Unix) at the beginning of the command line.

The interpreter may also source a few additional interactive startup files immediately before entering the interactive loop, unless the *`--norc`* option is specified. First .purerc in the user's home directory is read, then .purerc in the current working directory. These are ordinary Pure scripts which can be used to provide additional definitions for interactive usage. Finally, a .pure file in the current directory (containing a dump from a previous interactive session) is loaded if it is present.

When the interpreter is in interactive mode and reads from a tty, unless the *`--noediting`* option is specified, commands are usually read using **readline** or some compatible replace-

ment, providing completion for all commands listed under Interactive Usage, as well as for symbols defined in the running program. When exiting the interpreter, the command history is stored in ~/.pure_history, from where it is restored the next time you run the interpreter.

The interpreter also provides a simple source level debugger when run in interactive mode, see Debugging for details. To enable the debugger, you need to specify the `-g` option when invoking the interpreter. This option causes your script to run *much* slower, so you should only use this option if you want to run the debugger.

1.2.6 Verbosity and Debugging Options

The `-v` option is useful for debugging the interpreter, or if you are interested in the code your program gets compiled to. The level argument is optional; it defaults to 1. Seven different levels are implemented at this time. Only the first two levels will be useful for the average Pure programmer; the remaining levels are mostly intended for maintenance purposes.

1 (0x1, 001) denotes echoing of parsed definitions and expressions.

2 (0x2, 002) adds special annotations concerning local bindings (de Bruijn indices, subterm paths; this can be helpful to debug tricky variable binding issues).

4 (0x4, 004) adds descriptions of the matching automata for the left-hand sides of equations (you probably want to see this only when working on the guts of the interpreter).

8 (0x8, 010) dumps the "real" output code (LLVM assembler, which is as close to the native machine code for your program as it gets; you definitely don't want to see this unless you have to inspect the generated code for bugs or performance issues).

16 (0x10, 020) adds debugging messages from the bison(1) parser; useful for debugging the parser.

32 (0x20, 040) adds debugging messages from the flex(1) lexer; useful for debugging the lexer.

64 (0x40, 0100) turns on verbose batch compilation; this is useful if you want to see exactly which commands get executed during batch compilation (`-c`).

These values can be or'ed together, and, for convenience, can be specified in either decimal, hexadecimal or octal. Thus 0xff or 0777 always gives you full debugging output (which isn't likely to be used by anyone but the Pure developers). Some useful flag combinations for experts are (in octal) 007 (echo definitions along with de Bruijn indices and matching automata), 011 (definitions and assembler code), 021 (parser debugging output along with parsed definitions) and 0100 (verbose batch compilation).

Note that the `-v` option is only applied after the prelude has been loaded. If you want to debug the prelude, use the `-n` option and specify the prelude.pure file explicitly on the command line. Verbose output is also suppressed for modules imported through a `using` clause. As a remedy, you can use the interactive `show` command (see the Interactive Usage section) to list definitions along with additional debugging information.

1.2.7 Compilation Options

Besides the options listed above, the interpreter also understands some additional command line switches and corresponding environment variables to control various compilation options.

Code Generation Options

These options take the form `--opt` and `--noopt`, respectively, where `opt` denotes the option name (see below for a list of supported options). By default, these options are all enabled; `--noopt` disables the option, `--opt` reenables it. In addition, for each option `opt` there is also a corresponding environment variable `PURE_NOOPT` (with the option name in uppercase) which, when set, disables the option by default. (Setting this variable to any value will do, the interpreter only checks whether the variable exists in the environment.)

For instance, the `checks` option controls stack and signal checks. Thus `--nochecks` on the command line disables the option, and setting the `PURE_NOCHECKS` environment variable makes this the default, in which case you can use `--checks` on the command line to reenable the option.

Each code generation option can also be used as a **pragma** (compiler directive) in source code so that you can control it on a per-rule basis. The pragma must be on a line by itself, starting in column 1, and takes the following form (using `--nochecks` as an example):

```
#! --nochecks // line-oriented comment may go here
```

Currently, the following code generation options are recognized:

`--checks`
`--nochecks`
: Enable or disable various extra stack and signal checks. By default, the interpreter checks for stack overflows (if the `PURE_STACK` environment variable is set) and pending signals on entry to every function, see Stack Size and Tail Recursion and Handling of Asynchronous Signals for details. This is needed to catch these conditions in a reliable way, so we recommend to leave this enabled. However, these checks also make programs run a little slower (typically some 5%, YMMV). If performance is critical then you can disable the checks with the *`--nochecks`* option. (Even then, a minimal amount of checking will be done, usually on entry to every global function.)

`--const`
`--noconst`
: Enable or disable the precomputing of constant values in batch compilation (cf. Compiling Scripts). If enabled (which is the default), the values of constants in `const` definitions are precomputed at compile time (if possible) and then stored in the generated executable. This usually yields faster startup times but bigger executables. You can disable this option with *`--noconst`* to get smaller executables at the expense of slower startup times. Please see the Batch Compilation section for an example.

`--fold`

--nofold
: Enable or disable constant folding in the compiler frontend. This means that constant expressions involving int and double values and the usual arithmetic and logical operations on these are precomputed at compile time. (This is mostly for cosmetic purposes; the LLVM backend will perform this optimization anyway when generating machine code.) For instance:

```
> foo x = 2*3*x;
> show foo
foo x = 6*x;
```

Disabling constant folding in the frontend causes constant expressions to be shown as you entered them:

```
> #! --nofold
> bar x = 2*3*x;
> show bar
bar x = 2*3*x;
```

The same option also determines the handling of type aliases at compile time, see Type Rules.

--tc
--notc
: Enable or disable tail call optimization (TCO). TCO is needed to make tail-recursive functions execute in constant stack space, so we recommend to leave this enabled. However, at the time of this writing LLVM's TCO support is still bug-ridden on some platforms, so the *--notc* option allows you to disable it. (Note that TCO can also be disabled when compiling the Pure interpreter, in which case these options have no effect; see the *installation instructions* for details.)

Note: All of the options above also have a corresponding "option symbol" so that they can be queried and set using the facilities described under Conditional Compilation below. (The symbol is just the name of the option, e.g., `checks` for the *--checks*, *--nochecks* option and pragma.)

Besides these, there are the following special pragmas affecting the evaluation of some global function or macro, which is specified in the pragma. These pragmas can only be used in source code, they cannot be controlled using command line options or environment variables. Note that the given symbol `fun` may in fact be an arbitrary symbol (not just an identifier), so that these pragmas can also be applied to special operator symbols (cf. Lexical Matters). Also note that each of these pragmas also implicitly declares the symbol, so if a symbol needs any special attributes then it must be declared before any pragmas involving it (cf. Symbol Declarations).

--eager `fun`
: Instruct the interpreter to JIT-compile the given function eagerly. This means that native code will be created for the function, as well as all other (global or local) functions that may be called by the compiled function, as soon as the function gets recompiled.

This avoids the hiccups you get when a function is compiled on the fly if it is run for the first time, which is particularly useful for functions which are to be run in realtime (typically in multimedia applications). Please note that, in difference to the `--eager-jit` option, this feature is available for all LLVM versions (it doesn't require LLVM 2.7 or later).

--required fun
: Inform the batch compiler (cf. Compiling Scripts) that the given function symbol `fun` should never be stripped from the program. This is useful, e.g., if a function is never called explicitly but only through `eval`. Adding a `--required` pragma for the function then makes sure that the function is always linked into the program. Please see the Batch Compilation section for an example.

--defined fun
--nodefined fun
: These pragmas change the behaviour of functions defined in a Pure program. Pure's default mode is to evaluate function applications in a symbolic fashion using the equations (rewriting rules) supplied by the programmer, cf. Definitions and Expression Evaluation. This means that it is *not* normally an error if there is no equation which applies to the given function application to be evaluated; rather, the application simply becomes a "normal form" which stands for itself. E.g., here's what you get if you try to add an (undefined) symbol and a number:

```
> a+1;
a+1
```

The `--defined` pragma allows you to declare a function symbol as a "defined" function, so that it will raise a proper exception when no equation is applicable:

```
> #! --defined +
> a+1;
<stdin>, line 3: unhandled exception 'failed_match' while evaluating 'a+1'
```

The `--defined` status of a function can be changed at any time (causing the function to be recompiled on the fly if necessary), and the `--nodefined` pragma restores the default behaviour of returning a normal form upon failure:

```
> #! --nodefined +
> a+1;
a+1
```

More information and examples for common uses of the `--defined` pragma can be found under Defined Functions in the Caveats and Notes section.

--quoteargs fun
: This pragma tells the macro evaluator (cf. Macros) that the given macro should receive its arguments unevaluated, i.e., in quoted form. This is described in more detail in the Built-in Macros and Special Expressions section.

Conditional Compilation

As of version 0.49, Pure also provides a rudimentary facility for denoting optional and alternative code paths. This is supposed to cover the most common cases where conditional compilation is needed. (For more elaborate needs you can always use real Pure code which enables you to configure your program at runtime using, e.g., the `eval` function.)

Pure's conditional compilation pragmas are based on the notion of user-defined symbols (which can be really any text that does not contain whitespace or any of the shell wildcard characters `*?[]`) called compilation **options**. By default, all options are *undefined* and *enabled*. An option becomes *defined* as soon as it is set explicitly, either with an environment variable or one of the `--enable` and `--disable` pragmas, see below.

You can define the value of an option by setting a corresponding environment variable `PURE_OPTION_OPT`, where `OPT` is the option symbol in uppercase. The value of the environment variable should either be `0` (disabled) or `1` (enabled).

Options can be enabled and disabled in Pure scripts with the following pragmas, which are also available as command line options when invoking the Pure interpreter:

`--enable` `option`
`--disable` `option`
: Enable or disable the given option, respectively. Note that an option specified in the environment is overridden by a value specified with these options on the command line, which in turn is overridden by a corresponding pragma in source code.

The actual conditional compilation pragmas work in pretty much the same fashion as the C preprocessor directives `#if`, `#ifdef` etc. (except that, as already mentioned, an option is always *enabled* if it is undefined).

`--ifdef` `option`
`--ifndef` `option`
: Begins a code section which should be included in the program if the given option is defined or undefined, respectively.

`--if` `option`
`--ifnot` `option`
: Begins a code section which should be included in the program if the given option is enabled or disabled, respectively.

`--else`
: Begins an alternative code section which is included in the program if the corresponding `--ifdef`, `--ifndef`, `--if` or `--ifnot` section was excluded, and vice versa.

`--endif`
: Ends a conditional code section.

Conditional code sections may be nested to an arbitrary depth. Each `--ifdef`, `--ifndef`, `--if` or `--ifnot` pragma must be followed by a matching `--endif`. The `--else` section is optional; if present, it applies to the most recent `--ifdef`, `--ifndef`, `--if` or `--ifnot` section

not terminated by a matching `--endif`. Unmatched conditional pragmas warrant an error message by the compiler.

Conditional code is handled at the level of the lexical analyzer. Excluded code sections are treated like comments, i.e., the parser never gets to see them.

The `--ifdef` and `--ifndef` pragmas are typically used to change the default of an option without clobbering defaults set by the user through an environment variable or a command line option. For instance:

```
#! --ifndef opt
#! --disable opt
#! --endif
```

Here's a (rather contrived) example which shows all these pragmas in action. You may want to type this in the interpreter to verify that the code sections are indeed included and excluded from the Pure program as indicated:

```
// disable the 'bar' option
#! --disable bar

#! --ifdef foo
1/2; // excluded
#! --endif
#! --ifndef bar
1/3; // excluded
#! --endif

#! --if foo
foo x = x+1; // included
#! --if bar
bar x = x-1; // excluded
#! --else
bar x = x/2; // included
#! --endif // bar
#! --endif // foo

// reenable the 'bar' option
#! --enable bar

#! --if bar
bar 99; // included
#! --endif // bar

#! --ifnot foo
baz x = 2*x; // excluded
#! --endif // not foo
```

A few options are always predefined as "builtins" by the interpreter. This includes all of the options described under Code Generation Options and Warning Options, so that these can also be queried with `--if`, `--ifnot` and set with `--enable`, `--disable`. For instance:

```
#! --ifnot checks
puts "This program uses deep recursion, so we enable stack checks here!";
#! --enable checks
#! --endif // not checks

#! --if warn
puts "Beware of bugs in the above code.";
puts "I have only proved it correct, not tried it.";
#! --endif // warn
```

Moreover, the following options are provided as additional builtins which are useful for handling special compilation requirements as well as system and version dependencies.

- The `compiled` option is enabled if a program is batch-compiled. This lets you pick alternative code paths depending on whether a script is compiled to a native executable or not. Please see the example at the end of the Batch Compilation section for details.
- The `interactive` and `debugging` options are enabled if a program runs in interactive (`-i`) and/or debugging (`-g`) mode, respectively. These options are read-only; they cannot be changed with `--enable`, `--disable`. Example:

  ```
  #! --if interactive
  puts "Usage: run 'main filename'";
  #! --else
  main (argv!1);
  #! --endif
  ```

- The `version-x.y` option indicates a check against the version of the host Pure interpreter. `x.y` indicates the required (major/minor) version. You can also use `x.y+` to indicate version `x.y` or later, or `x.y-` for version `x.y` or earlier. By combining these, you can pick code depending on a particular range of Pure versions, or you can reverse the test to check for anything later or earlier than a given version:

  ```
  #! --if version-0.36+
  #! --if version-0.48-
  // code to be executed for Pure versions 0.36..0.48 (inclusive)
  #! --endif
  #! --endif

  #! --ifnot version-0.48-
  // code to be executed for Pure versions > 0.48
  #! --endif
  ```

- Last but not least, the interpreter always defines the target triplet of the host system as an option symbol. This is the same as what `sysinfo` returns, so you can check for a specific system like this:

  ```
  #! --if x86_64-unknown-linux-gnu
  // 64 bit Linux-specific code goes here
  #! --endif
  ```

  It goes without saying that this method isn't very practical if you want to check for a

wide range of systems. As a remedy, the `--if` and `--ifnot` pragmas treat shell glob patterns in tests for option symbols in a special way, by matching the pattern against the host triplet to see whether the condition holds. This allows you to write a generic test, e.g., for Windows systems like this:

```
#! --if *-mingw32
// Windows-specific code goes here
#! --endif
```

Warning Options

The `-w` option enables some additional warnings which are useful to check your scripts for possible errors. In particular, it will report implicit declarations of function and type symbols, which might indicate undefined or mistyped symbols that need to be fixed, see Symbol Lookup and Creation for details.

This option can also be controlled on a per-rule basis by adding the following pragmas to your script:

`--warn`
`--nowarn`
: Enable or disable compiler warnings. The `-w` flag sets the default for these pragmas.

`--rewarn`
: Reset compiler warnings to the default, as set with the `-w` flag (or not).

The latter pragma is useful to enable or disable warnings in a section of code and reset it to the default afterwards:

```
#! --warn
// Code with warnings goes here.
#! --rewarn
```

(The same could also be achieved with conditional compilation, but only much more clumsily. However, note that `--rewarn` only provides a single level of "backup", so nesting such sections is not supported.)

1.2.8 Startup Files

The interpreter may source various files during its startup. These are:

`~/.pure_history`
: Interactive command history.

`~/.purerc, .purerc, .pure`
: Interactive startup files. The latter is usually a dump from a previous interactive session.

prelude.pure
Standard prelude. If available, this script is loaded before any other definitions, unless *-n* was specified.

1.2.9 Environment

Various aspects of the interpreter can be configured through the following shell environment variables:

CC
CXX
C and C++ compiler used by the Pure batch compiler (`pure -c`) to compile and link native executables. Defaults to **gcc** and **g++**, respectively.

BROWSER
If the PURE_HELP variable is not set (see below), this specifies a colon-separated list of browsers to try for reading the online documentation. See http://catb.org/~esr/BROWSER/.

PURELIB
Directory to search for library scripts, including the prelude. If PURELIB is not set, it defaults to some location specified at installation time.

PURE_EAGER_JIT
Enable eager JIT compilation (same as *--eager-jit*), see Compiling Scripts for details.

PURE_ESCAPE
If set, interactive commands are prefixed with the first character in the value of this variable (same as *--escape*), see Interactive Usage for details.

PURE_HELP
Command used to browse the Pure manual. This must be a browser capable of displaying html files. Default is **w3m**.

PURE_INCLUDE
Additional directories (in colon-separated format) to be searched for included scripts.

PURE_LIBRARY
Additional directories (in colon-separated format) to be searched for dynamic libraries.

PURE_MORE
Shell command to be used for paging through output of the `show` command, when the interpreter runs in interactive mode. PURE_LESS does the same for evaluation results printed by the interpreter.

PURE_PS
Command prompt used in the interactive command loop ("> " by default).

PURE_STACK
Maximum stack size in kilobytes (default: 0 = unlimited).

Besides these, the interpreter also understands a number of other environment variables for setting various compilation options (see Compilation Options above) and commands to invoke different LLVM compilers on inline code (see Inline Code).

1.3 Pure Overview

Pure is a fairly simple yet powerful language. Programs are basically collections of term rewriting rules, which are used to reduce expressions to **normal form** in a symbolic fashion. For convenience, Pure also offers some extensions to the basic term rewriting calculus, like global variables and constants, nested scopes of local function and variable definitions, anonymous functions (lambdas), exception handling and a built-in macro facility. These are all described below and in the following sections.

Most basic operations are defined in the standard *prelude*. This includes the usual arithmetic and logical operations, as well as the basic string, list and matrix functions. The prelude is always loaded by the interpreter, so that you can start using the interpreter as a sophisticated kind of desktop calculator right away. Other useful operations are provided through separate library modules. Some of these, like the system interface and the container data structures, are distributed with the interpreter, others are available as separate add-on packages from the Pure website. A (very) brief overview of some of the modules distributed with the Pure interpreter can be found in the Standard Library section.

In this section we first give a brief overview of the most important elements of the Pure language. After starting out with a discussion of the lexical syntax, we proceed by explaining definitions and expressions, which are the major ingredients of Pure programs. After studying this section you should be able to write simple Pure programs. Subsequent sections then describe the concepts and notions introduced here in much greater detail and also cover the more advanced language elements which we only gloss over here.

1.3.1 Lexical Matters

Pure is a **free-format** language, i.e., whitespace is insignificant (unless it is used to delimit other symbols). Thus, in contrast to "layout-based" languages like Haskell, you *must* use the proper delimiters (`;`) and keywords (`end`) to terminate definitions and block structures. In particular, definitions and expressions at the toplevel have to be terminated with a semicolon, even if you're typing them interactively in the interpreter.

Comments use the same syntax as in C++: `//` for line-oriented, and `/* ... */` for multiline comments. The latter must not be nested. Lines beginning with `#!` are treated as comments, too; as already discussed above, on Unix-like systems this allows you to add a "shebang" to your main script in order to turn it into an executable program.

A few ASCII symbols are reserved for special uses, namely the semicolon, the "at" symbol `@`, the equals sign `=`, the backslash `\`, the Unix pipe symbol `|`, parentheses `()`, brackets `[]` and curly braces `{}`. (Among these, only the semicolon is a "hard delimiter" which is always a

lexeme by itself; the other symbols can be used inside operator symbols.) Moreover, there are some keywords which cannot be used as identifiers:

```
case   const      def     else       end     extern     if
infix  infixl     infixr  interface  let     namespace  nonfix
of     otherwise  outfix  postfix    prefix  private    public
then   type       using   when       with
```

Pure fully supports the **Unicode** character set or, more precisely, UTF-8. This is an ASCII extension capable of representing all Unicode characters, which provides you with thousands of characters from most of the languages of the world, as well as an abundance of special symbols for almost any purpose. If your text editor supports the UTF-8 encoding (most editors do nowadays), you can use all Unicode characters in your Pure programs, not only inside strings, but also for denoting identifiers and special operator symbols.

The customary notations for identifiers, numbers and strings are all provided. In addition, Pure also allows you to define your own operator symbols. Identifiers and other symbols are described by the following grammar rules in EBNF format:

```
symbol      ::=   identifier | special
identifier  ::=   letter (letter | digit)*
special     ::=   punct+
letter      ::=   "A"|...|"Z"|"a"|...|"z"|"_"|...
digit       ::=   "0"|...|"9"
punct       ::=   "!"|"#"|"$"|"%"|"&"|...
```

Pure uses the following rules to distinguish "punctuation" (which may only occur in declared operator symbols) and "letters" (identifier constituents). In addition to the punctuation symbols in the 7 bit ASCII range, the following code points in the Unicode repertoire are considered as punctuation: U+00A1 through U+00BF, U+00D7, U+00F7, and U+20D0 through U+2BFF. This comprises the special symbols in the Latin-1 repertoire, as well as the Combining Diacritical Marks for Symbols, Letterlike Symbols, Number Forms, Arrows, Mathematical Symbols, Miscellaneous Technical Symbols, Control Pictures, OCR, Enclosed Alphanumerics, Box Drawing, Blocks, Geometric Shapes, Miscellaneous Symbols, Dingbats, Miscellaneous Mathematical Symbols A, Supplemental Arrows A, Supplemental Arrows B, Miscellaneous Mathematical Symbols B, Supplemental Mathematical Operators, and Miscellaneous Symbols and Arrows. This should cover almost everything you'd ever want to use in an operator symbol. All other extended Unicode characters are effectively treated as "letters" which can be used as identifier constituents. (Charts of all Unicode symbols can be found at the Code Charts page of the Unicode Consortium.)

The following are examples of valid identifiers: `foo`, `foo_bar`, `FooBar`, `BAR`, `bar99`. Case is significant in identifiers, so `Bar` and `bar` are distinct identifiers, but otherwise the case of letters carries no meaning. Special symbols consist entirely of punctuation, such as `::=`. These may be used as operator symbols, but have to be declared before they can be used (see Symbol Declarations).

Pure also has a notation for qualified symbols which carry a namespace prefix. These take the following format (note that no whitespace is permitted between the namespace prefix

and the symbol):

```
qualified_symbol      ::=   [qualifier] symbol
qualified_identifier  ::=   [qualifier] identifier
qualifier             ::=   [identifier] "::"  (identifier "::")*
```

Example: `foo::bar`.

Number literals come in three flavours: integers, bigints (denoted with an `L` suffix) and floating point numbers (indicated by the presence of the decimal point and/or a base 10 scaling factor). Integers and bigints may be written in different bases (decimal, binary, octal and hexadecimal), while floating point numbers are always denoted in decimal.

```
number     ::=   integer | integer "L" | float
integer    ::=   digit+
                 | "0" ("X"|"x") hex_digit+
                 | "0" ("B"|"b") bin_digit+
                 | "0" oct_digit+
oct_digit  ::=   "0"|...|"7"
hex_digit  ::=   "0"|...|"9"|"A"|...|"F"|"a"|...|"f"
bin_digit  ::=   "0"|"1"
float      ::=   digit+ ["."  digit+] exponent
                 | digit* "."  digit+ [exponent]
exponent   ::=   ("E"|"e") ["+"|"-"] digit+
```

Examples: `4711`, `4711L`, `1.2e-3`. Numbers in different bases: `1000` (decimal), `0x3e8` (hexadecimal), `01750` (octal), `0b1111101000` (binary).

String literals are arbitrary sequences of characters enclosed in double quotes, such as `"Hello, world!"`.

```
string  ::=   '"' char* '"'
```

Special escape sequences may be used to denote double quotes and backslashes (`\"`, `\\`), control characters (`\b`, `\f`, `\n`, `\r`, `\t`, these have the same meaning as in C), and arbitrary Unicode characters given by their number or XML entity name (e.g., `\169`, `\0xa9` and `\&copy;` all denote the Unicode copyright character, code point U+00A9). As indicated, numeric escapes can be specified in any of the supported bases for integer literals. For disambiguating purposes, these can also be enclosed in parentheses. E.g., `"\(123)4"` is a string consisting of the character `\123` followed by the digit `4`. Strings can also be continued across line ends by escaping the line end with a backslash. The escaped line end is ignored (use `\n` if you need to embed a newline in a string). For instance,

```
"Hello, \
world.\n"
```

denotes the same string literal as

```
"Hello, world.\n"
```

1.3.2 Definitions and Expression Evaluation

The real meat of a Pure program is in its definitions. In Pure these generally take the form of equations which tell the interpreter how expressions are to be evaluated. For instance, the following two equations together define a function `fact` which computes, for each given integer `n`, the factorial of `n`:

```
fact 0 = 1;
fact n::int = n*fact (n-1) if n>0;
```

The first equation covers the case that `n` is zero, in which case the result is `1`. The second equation handles the case of a positive integer. Note the `n::int` construct on the left-hand side, which means that the equation is restricted to (machine) integers `n`. This construct is also called a "type tag" in Pure parlance. In addition, the `n>0` in the condition part of the second equation ensures that `n` is positive. If these conditions are met, the equation becomes applicable and we recursively compute `fact (n-1)` and multiply by `n` to obtain the result. The `fact` function thus computes the product of all positive integers up to `n`, which is indeed just how the factorial is defined in mathematics.

To give this definition a try, you can just enter it at the command prompt of the interpreter as follows:

```
> fact 0 = 1;
> fact n::int = n*fact (n-1) if n>0;
> fact 10;
3628800
```

On the surface, Pure is quite similar to other modern functional languages like Haskell and ML. But under the hood it is a much more dynamic language, more akin to Lisp. In particular, Pure is dynamically typed, so functions can process arguments of as many different types as you like. In fact, you can add to the definition of an existing function at any time. For instance, we can extend our example above to make the `fact` function work with floating point numbers, too:

```
> fact 0.0 = 1.0;
> fact n::double = n*fact (n-1) if n>0;
> fact 10.0;
3628800.0
```

Here we employed the constant `0.0` and the `double` type tag to define the factorial for the case of floating point numbers. Both `int` and `double` are built-in types of the Pure language. Our earlier definition for the `int` case still works as well:

```
> fact 10;
3628800
```

In FP parlance, we say that a function like `fact` is **polymorphic**, because it applies to different argument types. More precisely, the kind of polymorphism at work here is **ad-hoc**

polymorphism, because we have two distinct definitions of the same function which behave differently for different argument types.

Note that in this specific case, the two definitions are in fact very similar, to the point that the right-hand sides of the definitions are almost the same. Observing these similarities, we may also define `fact` in a completely generic way:

```
> clear fact
> fact n = 1 if n==0;
> fact n = n*fact (n-1) if n>0;
```

(Note that before we can enter the new definition, we first need to scratch our previous definition of `fact`, that's what the `clear fact` command does. This is necessary because, as we already saw, the interpreter would otherwise just keep adding equations to the definition of `fact` that we already have.)

Our new definition doesn't have any type tags on the left-hand side and will thus work with any type of numbers:

```
> fact 10; // int
3628800
> fact 30.0; // floating point
2.65252859812191e+32
> fact 50L; // bigint
30414093201713378043612608166064768844377641568960512000000000000L
```

Let's now take a look at how the equations are actually applied in the evaluation process. Conceptually, Pure employs term rewriting as its underlying model of computation, so the equations are applied as rewriting rules, reading them from left to right. An equation is applicable if its left-hand side matches the target term to be evaluated, in which case we can bind the variables in the left-hand side to the corresponding subterms in the target term. Equations are tried in the order in which they are written; as soon as the left-hand side of an equation matches (and the condition part of the equation, if any, is satisfied), it can be applied to reduce the target term to the corresponding right-hand side.

For instance, let's take a look at the target term `fact 3`. This matches both equations of our generic definition of `fact` from above, with `n` bound to `3`. But the condition `3==0` of the first equation fails, so we come to consider the second equation, whose condition `3>0` holds. Thus we can perform the reduction `fact 3 ==> 3*fact (3-1)` and then evaluate the new target term `3*fact (3-1)` recursively.

At this point, we have to decide which of the several subterms we should reduce first. This is also called the **reduction strategy** and there are different ways to go about it. For instance, we might follow the customary "call-by-value" strategy where the arguments of a function application are evaluated recursively before the function gets applied to it, and this is also what Pure normally does. More precisely, expressions are evaluated using the "leftmost-innermost" reduction strategy where the arguments are considered from left to right.

So this means that on the right-hand side of the second equation, first `n-1` (being the argument of `fact`) is evaluated, then `fact (n-1)` (which is an argument to the `*` operator), and finally `fact (n-1)` is multiplied by `n` to give the value of `fact n`. Thus the evaluation

of `fact 3` actually proceeds as follows (abbreviating reductions for the built-in arithmetic operations):

```
fact 3 => 3*fact 2 => 3*2*fact 1 => 3*2*1*fact 0 => 3*2*1*1 => 6.
```

We mention in passing here that Pure also has a few built-in "special forms" which take some or all of their arguments unevaluated, using "call by name" argument passing. This is needed to handle some constructs such as logical operations and conditionals in an efficient manner, and it also provides a way to implement "lazy" data structures. We'll learn about these later.

One of the convenient aspects of the rewriting model of computation is that it enables you to define a function by pattern matching on structured argument types. For instance, we might compute the sum of the elements of a list as follows:

```
> sum [] = 0;
> sum (x:xs) = x+sum xs;
```

This discriminates over the different cases for the argument value which might either be the empty list `[]` or a non-empty list of the from `x:xs` where the variables `x` and `xs` refer to the head element and the rest of the list, respectively. (The '`:`' infix operator is Pure's way of writing Lisp's "cons"; this works the same as in other modern FPLs and is discussed in much more detail later.)

Let's give it a try:

```
> sum (1..10);
55
```

Note that `1..10` denotes the list of all positive integers up to 10 here, so we get the sum of the numbers 1 thru 10 as the result, which is indeed 55. (The '`..`' operation is provided in Pure's prelude, i.e., it is part of the standard library.)

Due to its term rewriting semantics, Pure actually goes beyond most other functional languages in that it can do symbolic evaluations just as well as "normal" computations:

```
> square x = x*x;
> square 4;
16
> square (a+b);
(a+b)*(a+b)
```

In fact, leaving aside the built-in support for some common data structures such as numbers and strings, all the Pure interpreter really does is evaluate expressions in a symbolic fashion, rewriting expressions using the equations supplied by the programmer, until no more equations are applicable. The result of this process is called a **normal form** which represents the "value" of the original expression. Moreover, there's no distinction between "defined" and "constructor" function symbols in Pure, so *any* function symbol or operator can be used *anywhere* on the left-hand side of an equation, and may act as a constructor symbol if it happens to occur in a normal form term. This enables you to work with algebraic rules like associativity and distributivity in a direct fashion:

```
> (x+y)*z = x*z+y*z; x*(y+z) = x*y+x*z;
> x*(y*z) = (x*y)*z; x+(y+z) = (x+y)+z;
> square (a+b);
a*a+a*b+b*a+b*b
```

The above isn't possible in languages like Haskell and ML which always enforce that only "pure" constructor symbols (without any defining equations) may occur as a subterm on the left-hand side of a definition; this is also known as the **constructor discipline**. Thus equational definitions like the above are forbidden in these languages. Pure doesn't enforce the constructor discipline, so it doesn't keep you from writing such symbolic rules if you need them.

Another way of looking at this is that Pure allows you to have **constructor equations**. For instance, the following equation makes lists automatically stay sorted:

```
> x:y:xs = y:x:xs if x>y;
> [13,7,9,7,1]+[1,9,7,5];
[1,1,5,7,7,7,9,9,13]
```

This isn't possible in Haskell and ML either because it violates the constructor discipline; since `':'` is a constructor it can't simultaneously be a defined function in these languages. Pure gives you much more freedom there.

This symbolic mode of evaluation is rather unusual outside of the realm of symbolic algebra systems, but it provides the programmer with a very flexible model of computation and is one of Pure's most distinguishing features. In some cases, however, the unevaluated normal forms may also become a nuisance since they may obscure possible programming errors. Therefore Pure provides a special *`--defined`* pragma (cf. Code Generation Options) which forces a function to be treated as a defined function, so that it becomes more like functions in traditional untyped languages such as Lisp and Python which raise an exception under such conditions. This is described in more detail under Defined Functions in the Caveats and Notes section.

Variables in Equations

Taking another look at the examples above, you might wonder how the Pure interpreter figures out what the parameters (a.k.a. "variables") in an equation are. This is quite obvious in rules involving just variables and special operator symbols, such as `(x+y)*z = x*z+y*z`. However, what about an equation like `foo (foo bar) = bar`? Since most of the time we don't declare any symbols in Pure, how does the interpreter know that `foo` is a literal function symbol here, while `bar` is a variable?

The answer is that the interpreter considers the different positions in the left-hand side expression of an equation. Basically, a Pure expression is just a tree formed by applying expressions to other expressions, with the atomic subexpressions like numbers and symbols at the leaves of the tree. (This is true even for infix expressions like `x+y`, since in Pure these are always equivalent to a function application of the form `(+) x y` which has the atomic subterms `(+)`, `x` and `y` at its leaves.)

Now the interpreter divides the leaves of the expression tree into "head" (or "function") and "parameter" (or "variable") positions based on which leaves are leftmost in a function application or not. Thus, in an expression like `f x y z`, `f` is in the head or function position, while `x`, `y` and `z` are in parameter or variable positions. (Note that in an infix expression like `x+y`, `(+)` is the head symbol, not `x`, as the expression is really parsed as `(+) x y`, see above.)

Identifiers in head positions are taken as literal function symbols by the interpreter, while identifiers in variable positions denote, well, variables. We also refer to this convention as the **head = function rule**. It is quite intuitive and lets us get away without declaring the variables in equations. (There are some corner cases not covered here, however. In particular, Pure allows you to declare special "nonfix" symbols, if you need a symbol to be recognized as a literal even if it occurs in a variable position. This is done by means of a `nonfix` declaration, see Symbol Declarations for details.)

1.3.3 Expression Syntax

Like in other functional languages, expressions are the central ingredient of all Pure programs. All computation performed by a Pure program consists in the evaluation of expressions, and expressions also form the building blocks of the equational rules which are used to define the constants, variables, functions and macros of a Pure program.

Typical examples of the different expression types are summarized in the following table. Note that lambdas bind most weakly, followed by the special `case`, `when` and `with` constructs, followed by conditional expressions (`if`-`then`-`else`), followed by the simple expressions. Operators are a part of the simple expression syntax, and are parsed according to their declared precedences and associativities (cf. Symbol Declarations). Function application binds stronger than all operators. Parentheses can be used to group expressions and override default precedences as usual.

Type	Example	Description
Block	`\x y->2*x-y`	anonymous function (lambda)
	`case f u of x,y = x+y end`	case expression
	`x+y when x,y = f u end`	local variable definition
	`f u with f (x,y) = x+y end`	local function definition
Conditional	`if x>0 then x else -x`	conditional expression
Simple	`x+y, -x, x mod y`	operator application
	`sin x, max a b`	function application
Primary	`4711, 4711L, 1.2e-3`	number
	`"Hello, world!\n"`	string
	`foo, x, (+)`	function or variable symbol
	`[1,2,3], (1,2,3)`	list and tuple
	`{1,2;3,4}`	matrix
	`[x,-y | x=1..n; y=1..m; x<y]`	list comprehension
	`{i==j | i=1..n; j=1..m}`	matrix comprehension

The formal syntax of expressions is as follows. (Note that the `rule` and `simple_rule` elements are part of the definition syntax, which is explained in the Rule Syntax section.)

```
expr          ::=   "\" prim_expr+ "->" expr
                  | "case" expr "of" rules "end"
                  | expr "when" simple_rules "end"
                  | expr "with" rules "end"
                  | "if" expr "then" expr "else" expr
                  | simple_expr
simple_expr   ::=   simple_expr op simple_expr
                  | op simple_expr
                  | simple_expr op
                  | application
application   ::=   application prim_expr
                  | prim_expr
prim_expr     ::=   qualified_symbol
                  | number
                  | string
                  | "(" op ")"
                  | "(" left_op right_op ")"
                  | "(" simple_expr op ")"
                  | "(" op simple_expr ")"
                  | "(" expr ")"
                  | left_op expr right_op
                  | "[" exprs "]"
                  | "{" exprs (";" exprs)* [";"] "}"
                  | "[" expr "|" simple_rules "]"
                  | "{" expr "|" simple_rules "}"
exprs         ::=   expr ("," expr)*
op            ::=   qualified_symbol
left_op       ::=   qualified_symbol
right_op      ::=   qualified_symbol
rules         ::=   rule (";" rule)* [";"]
simple_rules  ::=   simple_rule (";" simple_rule)* [";"]
```

Primary Expressions

The Pure language provides built-in support for machine integers (32 bit), bigints (implemented using GMP), floating point values (double precision IEEE 754) and character strings (UTF-8 encoded). These can all be denoted using the corresponding literals described in Lexical Matters. Truth values are encoded as machine integers; as you might expect, zero denotes *false* and any non-zero value *true*, and the prelude also provides symbolic constants `false` and `true` to denote these. Pure also supports generic C pointers, but these don't have a syntactic representation in Pure, except that the predefined constant `NULL` may be used to denote a generic null pointer; other pointer values need to be created with external C functions.

Together, these atomic types of expressions make up most of Pure's **primary expression**

syntax. Pure also provides built-in support for some types of "compound primaries" (lists, tuples and matrices). We also list these here since they are typically denoted in some kind of bracketed form, even though some related non-primary expression types such as `x:y` or `x,y` really belong to the simple expressions.

`Numbers: 4711, 4711L, 1.2e-3`
: The usual C notations for integers (decimal: `1000`, hexadecimal: `0x3e8`, octal: `01750`) and floating point values are all provided. Integers can also be denoted in base 2 by using the `0b` or `0B` prefix: `0b1111101000`. Integer constants that are too large to fit into machine integers are promoted to bigints automatically. Moreover, integer literals immediately followed by the uppercase letter `L` are always interpreted as bigint constants, even if they fit into machine integers. This notation is also used when printing bigint constants, to distinguish them from machine integers.

`Strings: "Hello, world!\n"`
: String constants are double-quoted and terminated with a null character, like in C. In contrast to C, strings are always encoded in UTF-8, and character escapes in Pure strings have a more flexible syntax (borrowed from the author's Q language) which provides notations to specify any Unicode character. Please refer to Lexical Matters for details.

`Function and variable symbols: foo, foo_bar, BAR, foo::bar`
: These consist of the usual sequence of letters (including the underscore) and digits, starting with a letter. Case is significant, thus `foo`, `Foo` and `FOO` are distinct identifiers. The '`_`' symbol, when occurring on the left-hand side of an equation, is special; it denotes the **anonymous variable** which matches any value without actually binding a variable. Identifiers can also be prefixed with a namespace identifier, like in `foo::bar`. (This requires that the given namespace has already been created, as explained under Namespaces in the Declarations section.)

`Operator symbols: +, ==, not`
: For convenience, Pure also provides you with a limited means to extend the syntax of the language with special operator symbols by means of a corresponding **fixity** declaration, as discussed in section Symbol Declarations. Besides the usual infix, prefix and postfix operators, Pure also provides outfix (bracket) and nonfix (nullary operator) symbols. (Nonfix symbols actually work more or less like ordinary identifiers, but the `nonfix` attribute tells the compiler that when such a symbol occurs on the left-hand side of an equation, it is always to be interpreted as a literal, cf. The "Head = Function" Rule.)

  Operator (and nonfix) symbols may take the form of an identifier or a sequence of punctuation characters, which may optionally be qualified with a namespace prefix. These symbols must always be declared before use. Once declared, they are always special, and can't be used as ordinary identifiers any more. However, like in Haskell, by enclosing an operator in parentheses, such as `(+)` or `(not)`, you can turn it into an ordinary function symbol.

`Lists: [x,y,z], x:xs`
: Pure's basic list syntax is the same as in Haskell, thus `[]` is the empty list and `x:xs` denotes a list with head element `x` and tail list `xs`. The infix constructor symbol '`:`' is

declared in the prelude. It associates to the right, so that `x:y:z` is the same as `x:(y:z)`. The usual syntactic sugar for list values in brackets is also provided, thus `[x,y,z]` is exactly the same as `x:y:z:[]`. (This kind of list value is also called a "proper" list. Pure also permits "improper" list values such as `1:2:3` with a non-list value in the tail. These aren't of much use as ordinary list values, but are frequently used in patterns or symbolic expressions such as `x:y` where the tail usually is a variable.)

Lists can be nested to an arbitrary depth. Also note that, in contrast to Haskell, lists are not required to be homogeneous, so in general they may contain an arbitary mix of element types. E.g., `[1,2.0,[x,y]]` is a three-element list consisting of an integer, a floating point number and a nested list containing two symbols.

Pure also provides a notation for arithmetic sequences such as `1..5`, which denotes the list `[1,2,3,4,5]`. Note the missing brackets; Pure doesn't use any special syntax for arithmetic sequences, the '`..`' symbol is just an ordinary infix operator declared and defined in the prelude. Sequences with arbitrary stepsizes can be written by denoting the first two sequence elements using the '`:`' operator, as in `1.0:1.2..3.0`. To prevent unwanted artifacts due to rounding errors, the upper bound in a floating point sequence is always rounded to the nearest grid point. Thus, e.g., `0.0:0.1..0.29` actually yields `[0.0,0.1,0.2,0.3]`, as does `0.0:0.1..0.31`.

`Tuples: (x,y,z)`

Pure's tuples are a flat variant of lists which are often used as aggregate function arguments and results when no elaborate hierarchical structure is needed. They are constructed using the infix "pairing" operator '`,`', for which the empty tuple `()` acts as a neutral element (i.e., `(),x` is just `x`, as is `x,()`). Pairs always associate to the right, meaning that `x,y,z = x,(y,z) = (x,y),z`, where `x,(y,z)` is the normalized representation. These rules imply that tuples can't be nested and that there are no "true" 1-tuples distinct from their single members; if you need this then you should use lists instead (cf. Splicing Tuples and Matrices).

Note that the parentheses are not really part of the tuple syntax in Pure, they're just used to group expressions. So `(x,y,z)` denotes just `x,y,z`. But since the '`,`' operator has a low precedence, the parentheses are often needed to include tuples in other contexts. In particular, the parentheses are required to set off tuple elements in lists and matrices. E.g., `[(1,2),3,(4,5)]` denotes a three element list consisting of two tuples and an integer.

Mathematically, Pure's notion of tuples corresponds to a **monoid** with an associative binary operation '`,`' and neutral element `()`. This is different from the usual definition of tuples in mathematical logic, which are nestable and correspond to Pure's notion of lists. So in Pure you can take your pick and use either flat tuples or nestable lists, whatever is most convenient for the problem at hand.

`Matrices: {1.0,2.0,3.0}, {1,2;3,4}, {cos t,-sin t;sin t,cos t}`

Pure also offers matrices, a kind of two-dimensional arrays, as a built-in data structure which provides efficient storage and element access. These work more or less like their Octave/MATLAB equivalents, but using curly braces instead of brackets. Component values may either be individual elements ("scalars") or submatrices which are combined to form a larger matrix, provided that all dimensions match up. Here, a scalar

is any expression which doesn't yield a matrix; these are considered to be 1x1 submatrices for the purpose of matrix construction. (Note that this "splicing" behaviour pertains to matrix construction only; nested matrix patterns are always matched literally.)

The comma arranges submatrices and scalars in columns, while the semicolon arranges them in rows. So, if both `x` and `y` are `n` x `m` matrices, then `{x,y}` becomes an `n` x `2*m` matrix consisting of all the columns of `x` followed by all the columns of `y`. Likewise, `{x;y}` becomes a `2*n` x `m` matrix (all the rows of `x` above of all rows of `y`). For instance, `{{1;3},{2;4}}` is another way to write the 2x2 matrix `{1,2;3,4}`. Row vectors are denoted as 1 x `n` matrices (`{1,2,3}`), column vectors as `n` x 1 matrices (`{1;2;3}`). More examples can be found in the Matrices and Vectors section.

Pure supports both numeric and symbolic matrices. The former are homogeneous arrays of double, complex double or (machine) int matrices, while the latter can contain any mixture of Pure expressions. Pure will pick the appropriate type for the data at hand. If a matrix contains values of different types, or Pure values which cannot be stored in a numeric matrix, then a symbolic matrix is created instead (this also includes the case of bigints, which are considered as symbolic values as far as matrix construction is concerned). Numeric matrices use an internal data layout that is fully compatible with the GNU Scientific Library (GSL), and can readily be passed to GSL routines via the C interface. (The Pure interpreter does not require GSL, however, so numeric matrices will work even if GSL is not installed.)

`Comprehensions: [x,y | x=1..n; y=1..m; x<y], {f x | x=1..n}`

Pure provides both list and matrix comprehensions as a convenient means to construct list and matrix values from a "template" expression and one or more "generator" and "filter" clauses. The former bind a pattern to values drawn from a list or matrix, the latter are just predicates determining which generated elements should actually be added to the result. Comprehensions are in fact just syntactic sugar for a combination of lambdas, conditional expressions and certain list and matrix operations, but they are often much easier to write.

Thus, for instance, `[f x | x=1..n]` is pretty much the same as `map f (1..n)`, while `[x | x=xs; x>0]` corresponds to `filter (>0) xs`. However, comprehensions are considerably more general in that they allow you to draw values from different kinds of aggregates including lists, matrices and strings. Also, matrix comprehensions alternate between row and column generation so that most common mathematical abbreviations carry over quite easily. Patterns can be used on the left-hand side of generator clauses as usual, and will be matched against the actual list or matrix elements; any unmatched elements are filtered out automatically, like in Haskell.

More details and examples can be found in the Examples section; in particular, see List Comprehensions and Matrices and Vectors.

Simple Expressions

The rest of Pure's expression syntax mostly revolves around the notion of function applications. For convenience, Pure also allows you to declare pre-, post-, out- and infix operator symbols, but these are in fact just syntactic sugar for function applications; see Symbol Declarations for details. Function and operator applications are used to combine primary expressions to compound terms, also referred to as **simple expressions**; these are the data elements which are manipulated by Pure programs.

As in other modern FPLs, function applications are written simply as juxtaposition (i.e., in "curried" form) and associate to the left. This means that in fact all functions only take a single argument. Multi-argument functions are represented as chains of single-argument functions. For instance, in `f x y = (f x) y` first the function `f` is applied to the first argument `x`, yielding the function `f x` which in turn gets applied to the second argument `y`. This makes it possible to derive new functions from existing ones using **partial applications** which only specify some but not all arguments of a function. For instance, taking the `max` function from the prelude as an example, `max 0` is the function which, for a given `x`, returns `x` itself if it is nonnegative and zero otherwise. This works because `(max 0) x = max 0 x` is the maximum of `0` and `x`.

One major advantage of having curried function applications is that, without any further ado, functions become first-class objects. That is, they can be passed around freely both as parameters and as function return values. Much of the power of functional programming languages stems from this feature.

Operator applications are written using prefix, postfix, outfix or infix notation, as the declaration of the operator demands, but are just ordinary function applications in disguise. As already mentioned, enclosing an operator in parentheses turns it into an ordinary function symbol, thus `x+y` is exactly the same as `(+) x y`. For convenience, partial applications of infix operators can also be written using so-called **operator sections**. A *left section* takes the form `(x+)` which is equivalent to the partial application `(+) x`. A *right section* takes the form `(+x)` and is equivalent to the term `flip (+) x`. (This uses the `flip` combinator from the prelude which is defined as `flip f x y = f y x`.) Thus `(x+) y` is equivalent to `x+y`, while `(+x) y` reduces to `y+x`. For instance, `(1/)` denotes the reciprocal and `(+1)` the successor function. (Note that, in contrast, `(-x)` always denotes an application of unary minus; the section `(+-x)` can be used to indicate a function which subtracts `x` from its argument.)

The common operator symbols like `+`, `-`, `*`, `/` etc. are all declared at the beginning of the prelude, see the *Pure Library Manual* for a list of these. Arithmetic and relational operators mostly follow C conventions. However, since `!`, `&` and `|` are used for other purposes in Pure, the logical and bitwise operations, as well as the negated equality predicates are named a bit differently: `~`, `&&` and `||` denote logical negation, conjunction and disjunction, while the corresponding bitwise operations are named `not`, `and` and `or`. Moreover, following these conventions, inequality is denoted `~=`. Also note that `&&` and `||` are special forms which are evaluated in short-circuit mode (see Special Forms below), whereas the bitwise connectives receive their arguments using call-by-value, just like the other arithmetic operations.

Special Expressions

Some special notations are provided for conditional expressions as well as anonymous functions (lambdas) and local function and variable definitions. The latter are also called **block expressions** since they introduce local bindings of variable and function symbols which may override other global or local bindings of these symbols. This gives rise to a kind of **block structure** similar to Algol-like programming languages. Please check Scoping Rules below for more information about this.

The constructs described here are called "special" because, in contrast to the other forms of expressions, they cannot occur in normal form terms as first-class values (at least not literally; there is an alternative quoted representation of special expressions, however, which *can* be manipulated with macros and functions for meta programming purposes, cf. Built-in Macros and Special Expressions).

`Conditional expressions: if x then y else z`
: Evaluates to `y` or `z` depending on whether `x` is "true" (i.e., a nonzero integer). A `failed_cond` exception is raised if the condition is not an integer.

`Lambdas: \x -> y`
: These denote anonymous functions and work pretty much like in Haskell. A lambda matches its argument against the left-hand side pattern `x` and then evaluates the right-hand side body `y` with the variables in `x` bound to their corresponding values. Pure supports multiple-argument lambdas (e.g, `\x y -> x*y`), as well as pattern-matching lambda abstractions such as `\(x,y) -> x*y`. A `failed_match` exception is raised if the actual arguments do not match the given patterns.

`Case expressions: case x of u = v; ... end`
: Matches an expression, discriminating over a number of different cases, similar to the Haskell `case` construct. The expression `x` is matched in turn against each left-hand side pattern `u` in the rule list, and the first pattern which matches `x` gives the value of the entire expression, by evaluating the corresponding right-hand side `v` with the variables in the pattern bound to their corresponding values. A `failed_match` exception is raised if the target expression doesn't match any of the patterns.

`When expressions: x when u = v; ... end`
: An alternative way to bind local variables by matching a collection of subject terms against corresponding patterns, similar to Aardappel's `when` construct. A single binding `x when u = v end` is equivalent to the lambda expression `(\u -> x) v` or the `case` expression `case v of u = x end`, so it matches `v` against the pattern `u` and evaluates `x` with the variables in `u` bound to their corresponding values (or raises a `failed_match` exception if `v` doesn't match `u`). However, a `when` clause may contain multiple definitions, which are processed from left to right, so that later definitions may refer to the variables in earlier ones. (This is exactly the same as several nested single definitions, with the first binding being the "outermost" one.)

`With expressions: x with u = v; ... end`
: Defines local functions. Like Haskell's `where` construct, but it can be used anywhere inside an expression (just like Aardappel's `where`, but Pure uses the keyword `with` which

better lines up with `case` and `when`). Several functions can be defined in a single `with` clause, and the definitions can be mutually recursive and consist of as many equations as you want. Local functions are applied in the same way as global ones, i.e., the argument patterns of each rule are matched against the actual function arguments and the first rule which matches has its right-hand side evaluated with the variables in the argument patterns bound to their corresponding values. If none of the rules match then the function application remains unevaluated (it becomes a normal form), so no exception is raised in this case. (This is in contrast to a lambda which otherwise is pretty much like a nameless local function defined by a single rule.)

The block constructs are similar to those available in most modern functional languages. In Pure these constructs are all implemented in terms of the basic term rewriting machinery, using lambda lifting to eliminate local functions, and the following equivalences which reduce lambdas as well as `case` and `when` expressions to special kinds of local functions or local function applications:

```
\x1 ... xn -> y
≡ f with f x1 ... xn = y; f _ ... _ = throw failed_match end

case x of y1 = z1; ...; yn = zn end
≡ f x with f y1 = z1; ...; f yn = zn; f _ = throw failed_match end

x when y = z end
≡ f z with f y = x; f _ = throw failed_match end

x when y1 = z1; ...; yn = zn end
≡ x when yn = zn end ... when y1 = z1 end
```

Note that by convention these constructs report a `failed_match` exception in case of argument mismatch. So they're treated like defined functions, which is somewhat at odds with the term rewriting semantics. This is done for convenience, however, so that the programmer doesn't have to deal with unevaluated applications of nameless block constructs in normal form terms. The case of named local functions is considered different because it effectively represents a local rewriting system which should be treated accordingly, in order to allow for symbolic evaluation.

1.3.4 Special Forms

As already mentioned, some operations are actually implemented as special forms which process some or all of their arguments using call-by-name.

`if x then y else z`
: The conditional expression is a special form with call-by-name arguments `y` and `z`; only one of the branches is actually evaluated, depending on the value of `x`.

x **&&** y
x **||** y
: The logical connectives evaluate their operands in **short-circuit mode**. Thus the second operand is passed by name and will only be evaluated if the first operand fails to

determine the value of the expression. For instance, x&&y immediately becomes false if x evaluates to false; otherwise y is evaluated to give the value of the expression. The built-in definitions of these operations work as if they were defined by the following equations (but note that the second operand is indeed passed by name):

```
x::int && y = if x then y else x;
x::int || y = if x then x else y;
```

Note that this isn't quite the same as in C, as the results of these operations are *not* normalized, i.e., they may return nonzero values other than 1 to denote "true". (This has the advantage that these operations can be implemented tail-recursively, see Stack Size and Tail Recursion.) Thus, if you need a normalized truth value then you'll have to make sure that either both operands are already normalized, or you'll have to normalize the result yourself. (A quick way to turn a machine int x into a normalized truth value is to compute ~~x or x~=0.)

Moreover, if the built-in definition fails because the first operand is not a machine int, then the second operand will be evaluated anyway and the resulting application becomes a normal form, which gives you the opportunity to extend these operations with your own definitions just like the other built-in operations. Note, however, that in this case the operands are effectively passed by value.

x **$$** y

The sequencing operator $$ evaluates its left operand, immediately throws the result away and then goes on to evaluate the right operand which gives the result of the entire expression. This operator is useful to write imperative-style code such as the following prompt-input interaction:

```
> using system;
> puts "Enter a number:" $$ scanf "%g";
Enter a number:
21
21.0
```

We mention in passing here that the same effect can be achieved with a when clause, which also allows you to execute a function solely for its side-effects and just ignore the return value:

```
> scanf "%g" when puts "Enter a number:" end;
Enter a number:
21
21.0
```

x **&**

The & operator does lazy evaluation. This is the only postfix operator defined in the standard prelude. It turns its operand into a kind of parameterless anonymous closure, deferring its evaluation. These kinds of objects are also commonly known as **thunks** or **futures**. When the value of a future is actually needed (during pattern-matching, or when the value becomes an argument of a C call), it is evaluated automatically and gets memoized, i.e., the computed result replaces the thunk so that it only has to be computed once.

Futures are useful to implement all kinds of lazy data structures in Pure, in particular: lazy lists a.k.a. streams. A **stream** is simply a list with a thunked tail, which allows it to be infinite. The Pure prelude defines many functions for creating and manipulating these kinds of objects; for further details and examples please Lazy Evaluation and Streams in the Examples section.

quote x
‘ x

This special form quotes an expression, i.e., `quote x` (or, equivalently, `'x`) returns just `x` itself without evaluating it. The prelude also provides a function `eval` which can be used to evaluate a quoted expression at a later time. For instance:

```
> let x = '(2*42+2^12); x;
2*42+2^12
> eval x;
4180.0
```

This enables some powerful metaprogramming techniques, which should be well familiar to Lisp programmers. However, there are some notable differences to Lisp's quote, please see The Quote in the Examples section for details and more examples.

1.3.5 Toplevel

At the toplevel, a Pure program basically consists of rewriting rules (which are used to define functions, macros and types), constant and variable definitions, and expressions to be evaluated:

```
script  ::=   item*
item    ::=   "let" simple_rule ";"
            | "const" simple_rule ";"
            | "def" macro_rule ";"
            | "type" type_rule ";"
            | rule ";"
            | expr ";"
```

These elements are discussed in more detail in the Rule Syntax section. Also, a few additional toplevel elements are part of the declaration syntax, see Declarations.

lhs = rhs;

Rewriting rules always combine a left-hand side pattern (which must be a simple expression) and a right-hand side (which can be any kind of Pure expression described above). The same format is also used in `with`, `when` and `case` expressions. In toplevel rules, `with` and `case` expressions, this basic form can also be augmented with a condition `if guard` tacked on to the end of the rule, where `guard` is an integer expression which determines whether the rule is applicable. Moreover, the keyword `otherwise` may be used to denote an empty guard which is always true (this is syntactic sugar to point out the "default" case of a definition; the interpreter just treats this as a comment). Pure also provides some abbreviations for factoring out common left-hand or

right-hand sides in collections of rules; see the Rule Syntax section for details.

`type lhs = rhs;`
A rule starting with the keyword `type` defines a type predicate. This works pretty much like an ordinary rewriting rule, except that only a single right-hand side is permitted (which may also be omitted in some cases) and the left-hand side may involve at most one argument expression; see the Type Rules section for details. There's also an alternative syntax which lets you define types in a more abstract way and have the compiler generate the type rules for you; this is described in the Interface Types section.

`def lhs = rhs;`
A rule starting with the keyword `def` defines a macro function. No guards or multiple right-hand sides are permitted here. Macro rules are used to preprocess expressions on the right-hand side of other definitions at compile time, and are typically employed to implement user-defined special forms and simple kinds of optimization rules. See the Macros section below for details and examples.

`let lhs = rhs;`
Binds every variable in the left-hand side pattern to the corresponding subterm of the right-hand side (after evaluating it). This works like a `when` clause, but serves to bind global variables occurring free on the right-hand side of other function and variable definitions.

`const lhs = rhs;`
An alternative form of `let` which defines constants rather than variables. (These are not to be confused with nonfix symbols which simply stand for themselves!) Like `let`, this construct binds the variable symbols on the left-hand side to the corresponding values on the right-hand side (after evaluation). The difference is that `const` symbols can only be defined once, and thus their values do not change during program execution. This also allows the compiler to apply some special optimizations such as constant folding.

`expr;`
A singleton expression at the toplevel, terminated with a semicolon, simply causes the given value to be evaluated (and the result to be printed, when running in interactive mode).

1.3.6 Scoping Rules

A few remarks about the scope of identifiers and other symbols are in order here. Special expressions introduce **local scopes** of functions and variables. Specifically, lambda expressions, as well as the left-hand sides of rules in `case`, `when` and `with` expressions, bind the variables in the patterns to their corresponding values. In addition, a `with` expression also binds function names to the corresponding functions defined by the rules given in the expression. In either case, these bindings are limited to the scope of the corresponding construct. Inside that scope they override other (global or local) definitions of the same symbols which may be present in the surrounding program code. This gives rise to a hierarchical **block structure**

where each occurrence of a symbol refers to the innermost definition of that symbol visible at that point of the program.

The precise scoping rules for the different constructs are as follows:

- `\x -> y`: The scope of the variables bound by the pattern `x` is the lambda body `y`.
- `case x of u = v; ... end`: The scope of the variables bound by the pattern `u` in each rule is the corresponding right-hand side `v`.
- `x when u = v; ... end`: The scope of the variables bound by the pattern `u` in each rule extends over the right-hand sides of all subsequent rules and the target expression `x`.
- `x with u = v; ... end`: The scope of the variables bound by the pattern `u` in each rule is the corresponding right-hand side `v`. In addition, the scope of the *function* names defined by the `with` clause (i.e., the head symbols of the rules) extends over the right-hand sides of all rules and the target expression `x`. Note that this allows local function definitions to be mutually recursive, since the right-hand side of each rule in the `with` clause may refer to any other function defined by the `with` clause.

Like most modern functional languages, Pure uses **lexical** or **static** binding for local functions and variables. What this means is that the binding of a local name is completely determined at compile time by the surrounding program text, and does not change as the program is being executed. In particular, if a function returns another (anonymous or local) function, the returned function captures the environment it was created in, i.e., it becomes a (lexical) **closure**. For instance, the following function, when invoked with a single argument `x`, returns another function which adds `x` to its argument:

```
> foo x = bar with bar y = x+y end;
> let f = foo 99; f;
bar
> f 10, f 20;
109,119
```

This works the same no matter what other bindings of `x` may be in effect when the closure is invoked:

```
> let x = 77; f 10, (f 20 when x = 88 end);
109,119
```

In contrast to local bindings, Pure's toplevel environment binds global symbols **dynamically**, so that the bindings can be changed easily at any time during an interactive session. This is mainly a convenience for interactive usage, but works the same no matter whether the source code is entered interactively or being read from a script, in order to ensure consistent behaviour between interactive and batch mode operation.

In particular, you can easily bind a global variable to a new value by just entering a corresponding `let` command. For instance, contrast the following with the local `bar` function from above which had the `x` value bound in the surrounding context:

```
> clear x
> bar y = x+y;
> bar 10, bar 20;
x+10,x+20
> let x = 99;
> bar 10, bar 20;
109,119
> let x = 77;
> bar 10, bar 20;
87,97
```

Observe how changing the value of the global `x` variable immediately affects the value computed by the global `bar` function. This works pretty much like global variables in imperative languages, but note that in Pure the value of a global variable can only be changed with a `let` command at the toplevel. Thus referential transparency is unimpaired; while the value of a global variable may change between different toplevel expressions, it will always take the same value in a single evaluation.

Similarly, you can also add new equations to an existing function at any time. The Pure interpreter will then automatically recompile the function as needed. For instance:

```
> fact 0 = 1;
> fact n::int = n*fact (n-1) if n>0;
> fact 10;
3628800
> fact 10.0;
fact 10.0
> fact 1.0 = 1.0;
> fact n::double = n*fact (n-1) if n>1;
> fact 10.0;
3628800.0
> fact 10;
3628800
```

In interactive mode, it is even possible to completely erase a function definition and redo it from scratch, see section Interactive Usage for details.

So, while the meaning of a local symbol never changes once its definition has been processed, toplevel definitions may well evolve while the program is being processed, and the interpreter will always use the latest definitions at a given point in the source when an expression is evaluated.

Note: As already mentioned, this behaviour makes Pure much more convenient to use in an interactive setting. We should point out, however, that dynamic environments are often frowned upon by functional programming purists (for good reasons), and Pure's dynamic toplevel certainly has its pitfalls just like any other. Specifically, even in a script file you'll have to take care that all symbols needed in an evaluation are completely defined before entering the expression to be evaluated. Nevertheless, it is expected that most Pure programmers will use Pure interactively most of the time, and so tailoring the design to interactive usage seems justifiable in this case.

1.4 Rule Syntax

Basically, the same rule syntax is used in all kinds of global and local definitions. However, some constructs (specifically, `when`, `let`, `const`, `type` and `def`) use a variation of the basic rule syntax which does away with guards and/or multiple left-hand or right-hand sides. The syntax of these elements is captured by the following grammar rules:

```
rule         ::=   pattern ("|" pattern)* "=" expr [guard]
                   (";" "=" expr [guard])*
type_rule    ::=   pattern ("|" pattern)* [ "=" expr [guard] ]
macro_rule   ::=   pattern ("|" pattern)* "=" expr
simple_rule  ::=   pattern "=" expr | expr
pattern      ::=   simple_expr
guard        ::=   "if" simple_expr
                   | "otherwise"
                   | guard "when" simple_rules "end"
                   | guard "with" rules "end"
```

When matching against a function or macro call, or the subject term in a `case` expression, the rules are always considered in the order in which they are written, and the first matching rule (whose guard evaluates to a nonzero value, if applicable) is picked. (Again, the `when` construct is treated differently, because each rule is actually a separate definition.)

1.4.1 Patterns

The left-hand side of a rule is a special kind of simple expression, called a **pattern**. The variables in a pattern serve as placeholders which are bound to corresponding values when the rule is applied to a target expression. To these ends, the pattern is **matched** against the target expression, i.e., the literal parts of the pattern are compared against the target expression and, if everything matches up, the variables in the pattern are **bound** to (set to the value of) the corresponding subterms of the target expression.

Patterns are pervasive in Pure; they are used on the left-hand side of function and macro definitions, just as well as in global and local variable definitions. For instance, the following variable definition matches the result of evaluating the right-hand side list expression against the pattern `x:y:xs` and binds the variables `x` and `y` to the first two elements of the resulting list and `xs` to the list of remaining elements, respectively. We can then place `x` and `y` at the end of the list, thereby performing a kind of "rotation" of the first two list members:

```
> let x:y:xs = 1..10;
> xs+[x,y];
[3,4,5,6,7,8,9,10,1,2]
```

The same works with a local variable definition:

```
> xs+[x,y] when x:y:xs = 1..10 end;
[3,4,5,6,7,8,9,10,1,2]
```

Or with a `case` expression:

```
> case 1..10 of x:y:xs = xs+[x,y] end;
[3,4,5,6,7,8,9,10,1,2]
```

The arguments of functions (and macros) are handled in the same fashion, too:

```
> rot2 (x:y:xs) = xs+[x,y];
> rot2 (1..10);
[3,4,5,6,7,8,9,10,1,2]
```

However, there is a big difference here. For global and local variable definitions, it is an error if the pattern does not match the target expression:

```
> let x:y:xs = [1];
<stdin>, line 7: failed match while evaluating 'let x:y:xs = [1]'
```

The same holds if the target expression doesn't match any of the left-hand side patterns in a `case` expression:

```
> case [1] of x:y:xs = xs+[x,y] end;
<stdin>, line 8: unhandled exception 'failed_match' while evaluating
'case [1] of x:y:xs = xs+[x,y] end'
```

(The error message is slightly different in this case, but the reported kind of exception is actually the same as with the `let` expression above.)

This doesn't normally happen with functions and macros. Instead, a match failure just means that the corresponding rule will be bypassed and other rules will be tried instead. If there are no more rules, the target expression becomes a normal form which is simply returned as is:

```
> rot2 [1];
rot2 [1]
```

This may come as a surprise (other functional languages will give you an error in such cases), but is a crucial feature of term rewriting languages, as it opens the door to symbolic evaluation techniques, see Definitions and Expression Evaluation.

There are two different ways to force a function definition to bail out with an error if you prefer that behaviour. First, you can provide an explicit rule which raises an exception (cf. Exception Handling). But this can make it difficult or even impossible to add more rules to the function later, as discussed below. Instead, you may want to use the *`--defined`* pragma as follows:

```
> #! --defined rot2
> rot2 [1];
<stdin>, line 13: unhandled exception 'failed_match' while evaluating 'rot2 [1]'
```

Note: This pragma tells the compiler that `rot2` is supposed to be a "defined" function, which means that it should be an error if no rule applies to it; please see Defined Functions in the Caveats and Notes section for details. Also note that exceptions will always interfere with symbolic evaluation and thus the use of this facility isn't really recommended. However, there are situations in which it can make your life a lot easier.

One of Pure's key features is that you can usually just keep on adding new rules to existing function definitions in order to handle different kinds of arguments. As already mentioned, the rules will then be considered in the order in which they are written, and the first rule which matches the given arguments will be used to reduce the function application. For instance, adding the following rule we can make the `rot2` function also work with tuples:

```
> rot2 (x,y,xs) = xs,x,y;
> rot2 (1,2,3,4,5);
3,4,5,1,2
```

This is also known as **ad-hoc polymorphism**. By these means, you can make a function apply to as many different kinds of arguments as you want, and the pattern matching handles the necessary "dispatching" so that the right rule gets invoked for the provided arguments.

Pattern matching is not limited to the predefined aggregates such as lists, tuples and matrices. In principle, any legal Pure expression can occur as a pattern on the left-hand side of a rule or definition, so you can also write something like:

```
> rot2 (point x y z) = point z x y;
> rot2 (point 1 2 3);
point 3 1 2
```

Or even:

```
> foo (foo x) = foo x;
> bar (foo x) = foo (bar x);
> foo (bar (foo 99));
foo (bar 99)
```

Note that symbolic rules like in the latter example (which in this case express the idempotence of `foo` and a kind of commutativity with respect to `bar`) often involve symbols which play the role of both a function *and* a constructor symbol.

Syntactically, patterns are simple expressions, thus special expressions need to be parenthesized if you want to include them in a pattern. (In fact, special expressions are given special treatment if they occur in patterns, see the Macros section for details.) A few other special elements in patterns are discussed below.

The "Head = Function" Rule

A central ingredient of all patterns are of course the variables which get bound in the pattern matching process. Pure is a rather terse language and thus it has no explicit way to declare

which identifiers are the variables. Instead, the compiler figures them out on its own, using a rather intuitive rule already explained in Variables in Equations.

Recall that the variables in a pattern are the identifiers in "variable positions". The **head = function** rule tells us that a variable position is any leaf (atomic subexpression) of the expression tree which is *not* the head symbol of a function application. Thus a pattern like `f (g x) y` contains the variables `x` and `y`, whereas `f` and `g` are interpreted as literal function symbols. This rule also applies to the case of infix, prefix or postfix operator symbols, if we write the corresponding application in its unsugared form. E.g., `x+y*z` is equivalent to `(+) x ((*) y z)` which contains the variables `x`, `y` and `z` and the literal function symbols `(+)` and `(*)`.

There are some exceptions to the "head = function" rule. Specifically, it is possible to declare an identifier as a `nonfix` symbol so that it will be interpreted as a literal function symbol even if it occurs in a variable position, see Symbol Declarations for details. For instance:

```
nonfix nil;
foo nil = 0;
```

Note that since `nil` is declared as a `nonfix` symbol here, the symbol is interpreted as a literal rather than a variable in the left-hand side `foo nil`, and thus `foo` will return `0` for a literal `nil` value only.

Another case which needs special consideration are patterns consisting of a single identifier, such as `x`. Here the meaning depends on the kind of construct. All variable-binding constructs (`let`, `const`, `when` and `case`) treat a singleton identifier as a variable (unless it is declared `nonfix`). Thus all of the following constructs will have the expected result of binding the variable `x` to the given list value `[1,2,3]`. In either case the result is `[0,1,2,3]`:

```
let x = [1,2,3]; 0:x;
0:x when x = [1,2,3] end;
case [1,2,3] of x = 0:x end;
```

In contrast, a single identifier is always interpreted as a literal if it occurs on the left-hand side of a function or macro definition, so that the following rule defines a parameterless function `y`:

```
y = [1,2,3]; 0:y;
```

(While they yield the same values here, there are some notable differences between the parameterless function `y` and the global variable `x` defined above; see Defining Functions for details.)

Please also check "Head = Function" Pitfalls in the Caveats and Notes section which has some some further interesting details and workarounds concerning the "head = function" rule.

Constant Patterns

Constants in patterns must be matched literally. For instance:

```
foo 0 = 1;
```

This will only match an application of `foo` to the machine integer `0`, not `0.0` or `0L` (even though these compare equal to `0` using the '`==`' operator).

The Anonymous Variable

The '`_`' symbol is special in patterns; it denotes the **anonymous variable** which matches an arbitrary value (independently for all occurrences) without actually binding a variable. This is useful if you don't care about an argument or one of its components, in which case you can just use the anonymous variable as a placeholder for the value and don't have to invent a variable name for it. For instance:

```
foo _ _ = 0;
```

This will match the application of `foo` to any combination of two arguments (and just ignore the values of these arguments).

Non-Linear Patterns and Syntactic Equality

In contrast to Haskell, patterns may contain repeated variables (other than the anonymous variable), i.e., they may be **non-linear**. Thus rules like the following are legal in Pure, and will only be matched if all occurrences of the same variable in the left-hand side pattern are matched to the same value:

```
> foo x x = x;
> foo 1 1;
1
> foo 1 2;
foo 1 2
```

Non-linear patterns are particularly useful for computer algebra where you will frequently encounter rules such as the following:

```
> x*y+x*z = x*(y+z);
> a*(3*4)+a*5;
a*17
```

The notion of "sameness" employed here is that of syntactical identity, which means that the matched subterms must be identical in structure and content. The prelude provides syntactic equality as a function `same` and a comparison predicate '`===`'. Thus the above definition of `foo` is roughly equivalent to the following:

```
foo x y = x if same x y;
```

It is important to note the differences between syntactic equality embodied by `same` and '`===`', and the "semantic" equality operator '`==`'. The former are always defined on all terms,

whereas '==' is only available on data where it has been defined explicitly, either in the prelude or by the programmer. Also note that '==' may assert that two terms are equal even if they are syntactically different. Consider, e.g.:

```
> 0==0.0;
1
> 0===0.0;
0
```

This distinction is actually quite useful. It gives the programmer the flexibility to define '==' in any way that he sees fit, which is consistent with the way the other comparison operators like '<' and '>' are handled in Pure.

Syntactic equality is also used in pattern matching in order to decide whether a constant in a pattern matches the corresponding subterm in the target expression. This explains why the pattern `foo 0`, as already mentioned, only matches an application of `foo` to the machine integer `0`, not `0.0` or `0L` which aren't syntactically equal to `0`.

However, there is one caveat here. Due to its term rewriting heritage, Pure distinguishes between literal function symbols in patterns and named functions. The latter are runtime objects which are only considered syntactically equal if they not only have the same name but actually refer to the same (global or local) closure. In contrast, a function symbol in a pattern is just a literal symbol without reference to any particular closure that the symbol may be bound to in some context. Thus a function symbol in a pattern matches *any* instance of the symbol in the target expression, no matter whether it happens to be a pure constructor, quoted symbol or any named closure bound to that symbol.

This leads to some discrepancies between pattern matching and syntactic equality which may be surprising at first sight. For instance, consider:

```
> foo x = case x of bar y = x===bar y end;
> bar x y = x+y;
> foo (bar 99);
1
> foo ('bar 99);
0
> foo (bar 99) with bar x y = x*y end;
0
```

Note that the argument term `bar 99` matches the pattern `bar y` in the `case` expression in either case, even though in the last two expressions `bar` is *not* considered syntactically equal to the global `bar` function because it is quoted (cf. The Quote) or bound to a local closure of the same name, respectively.

Special Patterns

Last but not least, patterns may also contain the following special elements which are not permitted in right-hand side expressions:

- A Haskell-style **"as" pattern** of the form *variable* `@` *pattern* binds the given variable to the expression matched by the subpattern *pattern* (in addition to the variables bound

by *pattern* itself). This is convenient if the value matched by the subpattern is to be used on the right-hand side of an equation.

- A left-hand side variable (including the anonymous variable) may be followed by a **type tag** of the form `::` *name*, where *name* is either one of the built-in type symbols `int`, `bigint`, `double`, `string`, `matrix`, `pointer`, or an identifier denoting a user-defined data type. The variable can then match only values of the designated type. Thus, for instance, '`x::int`' only matches machine integers. See the Type Tags section below for details.

To these ends, the expression syntax is augmented with the following grammar rule (but note that this form of expression is in fact only allowed on the left-hand side of a rule):

```
prim_expr  ::=   qualified_identifier
                 ("::"  qualified_identifier | "@" prim_expr)
```

As shown, both "as" patterns and type tags are primary expressions, and the subpattern of an "as" pattern is a primary expression, too. Thus, if a compound expression is to be used as the subpattern, it *must* be parenthesized. For instance, the following function duplicates the head element of a list:

```
foo xs@(x:_) = x:xs;
```

Note that if you accidentally forget the parentheses around the subpattern `x:_`, you still get a syntactically correct definition:

```
foo xs@x:_ = x:xs;
```

But this gets parsed as `(foo xs@x):_ = x:xs`, which is most certainly *not* what you want. It is thus a good idea to just always enclose the subpattern with parentheses in order to prevent such glitches.

Note: Another pitfall is that the notation `foo::bar` is also used to denote "qualified symbols" in Pure, cf. Namespaces. Usually this will be resolved correctly, but if `foo` happens to also be a valid namespace then most likely you'll get an error message about an undeclared symbol. You can always work around this by adding spaces around the '`::`' symbol, as in `foo :: bar`. Spaces are never permitted in qualified symbols, so this makes it clear that the construct denotes a type tag. The same applies if the variable or the tag is a qualified identifier; in this case they should always be separated by whitespace.

1.4.2 Type Tags

Like Lisp, Pure is essentially a typeless language and doesn't really have a built-in notion of "data types". Rather, all data belongs to the same universe of terms. However, for convenience it is possible to describe data domains by means of (unary) type *predicates* which may denote arbitrary sets of terms. The names of these type predicates can then be used as **type tags** on variables, so that they can only be matched by values of the given type.

We have to emphasize here that Pure's notion of types has nothing to do with static typing. Type tags are merely used at runtime to restrict the kind of data that can be matched by a rule (and by the compiler to generate better code in some cases). But they will never cause the compiler to impose a static typing discipline and spit out corresponding "type errors". (This wouldn't make any sense in Pure anyway, as failure to match any of the rules given in the definition of a function simply means that a function application is in normal form.)

Some basic types are built into the language. The corresponding tags enable you to match the built-in types of terms for which there is no way to spell out all "constructors", as there are infinitely many (or none, as in the case of `pointer` values which are constructed and inspected using special primitives, but are otherwise "opaque" at the Pure level). Specifically, the following data types are built-in (in fact, the pattern matcher has special knowledge about these so that they can be matched very efficiently):

type `int`
: The type of machine integers.

type `bigint`
: The type of arbitrary precision integers (GMP bigints).

type `double`
: The type of double precision floating point numbers.

type `string`
: The type of character strings.

type `matrix`
: The type of all numeric and symbolic matrix values.

type `pointer`
: The type of C pointer values.

Pure's standard library provides additional data types along with the corresponding operations, such as rational and complex numbers, lists, tuples and the container data types (sets, dictionaries, etc.). These are all described in the *Pure Library Manual*.

You can define your own data types using a special kind of rule syntax which is explained in Type Rules below. For instance, we might represent points in the plane using a constructor symbol `Point` which gets applied to pairs of coordinates. We can then define the `point` data type as follows:

```
type point (Point x y);
```

This introduces the type symbol `point` and specifies that this type consists of terms of the form `Point x y`. We can now equip this data type with an operation `point` to construct a point from its coordinates, two operations `xcoord` and `ycoord` to retrieve the coordinates, and an operation `move` to change the coordinates to the given values:

```
point x y = Point x y;
xcoord (Point x y) = x;
ycoord (Point x y) = y;
move (Point _ _) x y = Point x y;
```

Next we might define a function `translate` which shifts the coordinates of a point by a given amount in the x and y directions as follows:

```
translate x y p::point = move p (xcoord p+x) (ycoord p+y);
```

Note the use of `point` as a type tag on the `p` variable. By these means, we can ensure that the argument is actually an instance of the point data type we just defined. The type tag acts just like an extra guard of the equation defining `translate`, but all the necessary type checking is done automatically during pattern matching. This is often more convenient (and, depending on the implementation, the compiler may generate more efficient code for a type tag than for an ordinary guard).

The `translate` function can be invoked as follows:

```
> let p::point = point 3 3;
> p; translate 1 2 p;
Point 3 3
Point 4 5
```

One important point to note here is that `translate` can be defined without knowing or assuming *anything* about the internal representation of the `point` data type. We have defined `point` as a **concrete data type** in this example, making its constructor and internal structure visible in the rest of the program. This is often convenient, but the `Point` constructor might just as well be hidden by making it a private member of some namespace (cf. Namespaces), so that all accesses to the data structure would have to be done through the provided operations. Such a data type is also known as an **abstract data type** (ADT).

Note: As we've already seen, Pure has some powerful capabilities which enable you to write functions to inspect and manipulate terms in a completely generic fashion. Thus the internal structure of term data is never truly opaque in Pure and it is always possible to break the "abstraction barrier" provided by an ADT. But if the user of an ADT plays such dirty tricks to wreak havoc on the internal representation of an ADT, he gets what he deserves.

Pure provides some additional facilities to ease the handling of abstract data types. Specifically, instead of defining `point` as a concrete data type using a `type` rule, we might also specify it as an **interface type** which merely lists the supported operations as follows:

```
interface point with
  xcoord p::point;
  ycoord p::point;
  move p::point x y;
end;
```

We can implement this type the same way as before:

```
point x y = Point x y;
xcoord (Point x y) = x;
ycoord (Point x y) = y;
move (Point _ _) x y = Point x y;
```

The definition of the `translate` function is also unchanged:

```
translate x y p::point = move p (xcoord p+x) (ycoord p+y);
```

The difference is that now the structure of members of the type is not made explicit *anywhere* in the definition of the type. Instead, the compiler figures out which data matches the `point` tag on its own. We can check the actual term patterns making up the `point` type with the `show interface` command:

```
> show interface point
type point (Point x y);
```

As you can see, the compiler derived our previous definition of the type. But in fact `translate` will now work with *any* data type which implements the `point` interface (i.e., provides the `xcoord`, `ycoord` and `move` operations), so we may swap out the underlying data structure on a whim. For instance, if we'd like to use vectors instead of constructor terms, all we have to do is to provide a corresponding construction function and implement the interface operations:

```
vpoint x y = {x,y};
xcoord {x,y} = x;
ycoord {x,y} = y;
move {_,_} x y = {x,y};
```

After these definitions the new data representation works just fine with existing `point` operations such as `translate`:

```
> show interface point
type point (Point x y);
type point {x,y};
> let p::point = vpoint 3 3;
> p; translate (1,2) p;
{3,3}
{4,5}
```

This separation of interface and implementation of a data structure is an important ingredient of software engineering techniques. More examples and detailed explanations of Pure's notions of type predicates and interface types can be found in the Type Rules and Interface Types sections.

1.4.3 General Rules

The most general type of rule, used in function definitions and `case` expressions, consists of a left-hand side pattern, a right-hand side expression and an optional guard. The left-hand side of a rule can be omitted if it is the same as for the previous rule. This provides a convenient means to write out a collection of equations for the same left-hand side which discriminates over different conditions:

```
lhs       = rhs if guard;
          = rhs if guard;
```

```
        ...
        = rhs otherwise;
```

For instance:

```
fact n  = n*fact (n-1) if n>0;
        = 1 otherwise;
```

This expands to:

```
fact n  = n*fact (n-1) if n>0;
fact n  = 1 otherwise;
```

Pure also allows a collection of rules with different left-hand sides but the same right-hand side(s) to be abbreviated as follows:

```
lhs     |
        ...
lhs     = rhs;
```

This is useful, e.g., if you specialize a rule to different type tags on the left-hand side variables. For instance:

```
fact n::int    |
fact n::double = n*fact(n-1) if n>0;
               = 1 otherwise;
```

This expands to:

```
fact n::int    = n*fact(n-1) if n>0;
               = 1 otherwise;
fact n::double = n*fact(n-1) if n>0;
               = 1 otherwise;
```

In fact, the left-hand sides don't have to be related at all, so you can also write something like:

```
foo x | bar y = x*y;
```

Which expands to:

```
foo x = x*y;
bar y = x*y;
```

But more often you'll have an "as" pattern which binds a common variable to a parameter value after checking that it matches one of several possible argument patterns (which is slightly more efficient than using an equivalent type-checking guard). E.g., the following definition binds the `xs` variable to the parameter of `foo`, which may be either the empty list or a list starting with an integer:

```
foo xs@[] | foo xs@(_::int:_) = bar xs;
```

The `|` notation also works in `case` expressions, which is convenient if different cases should be mapped to the same value, e.g.:

```
case ans of "y" | "Y" = 1; _ = 0; end;
```

Sometimes it is useful if local definitions (`when` and `with`) can be shared by the right-hand side and the guard of a rule. This can be done by placing the local definitions behind the guard, as follows (we only show the case of a single `when` clause here, but of course there may be any number of `when` and `with` clauses behind the guard):

```
lhs = rhs if guard when defns end;
```

Note that this is different from the following, which indicates that the definitions only apply to the guard but not the right-hand side of the rule:

```
lhs = rhs if (guard when defns end);
```

Conversely, definitions placed *before* the guard only apply to the right-hand side but not the guard (no parentheses are required in this case):

```
lhs = rhs when defns end if guard;
```

An example showing the use of a local variable binding spanning both the right-hand side and the guard of a rule is the following quadratic equation solver, which returns the (real) solutions of the equation `x^2+p*x+q = 0` if the discriminant `d = p^2/4-q` is nonnegative:

```
> using math;
> solve p q = -p/2+sqrt d,-p/2-sqrt d if d>=0 when d = p^2/4-q end;
> solve 4 2; solve 2 4;
-0.585786437626905,-3.41421356237309
solve 2 4
```

Note that the above definition leaves the case of a negative discriminant undefined.

1.4.4 Simple Rules

As already mentioned, `when`, `let` and `const` use a simplified kind of rule syntax which just consists of a left-hand and a right-hand side separated by the equals sign. In this case the meaning of the rule is to bind the variables in the left-hand side of the rule to the corresponding subterms of the value of the right-hand side. This is also called a **pattern binding**.

Guards or multiple left-hand or right-hand sides are not permitted in these rules. However, it is possible to omit the left-hand side if it is just the anonymous variable '_' by itself, indicating that you don't care about the result. The right-hand side is still evaluated, if only for its side-effects, which is handy, e.g., for adding debugging statements to your code. For instance, here is a variation of the quadratic equation solver which also prints the discriminant after it has been computed:

```
> using math, system;
> solve p q = -p/2+sqrt d,-p/2-sqrt d if d>=0
> when d = p^2/4-q; printf "The discriminant is: %g\n" d; end;
> solve 4 2;
The discriminant is: 2
```

```
-0.585786437626905,-3.41421356237309
> solve 2 4;
The discriminant is: -3
solve 2 4
```

Note that simple rules of the same form `lhs = rhs` are also used in macro definitions (`def`), to be discussed in the Macros section. In this case, however, the rule denotes a real rewriting rule, not a pattern binding, hence the left-hand side is mandatory in these rules.

1.4.5 Type Rules

In Pure the definition of a type takes a somewhat unusual form, since it is not a static declaration of the structure of the type's members, but rather an arbitrary predicate which determines through a runtime check which terms belong to the type. Thus the definition of a type looks more like an ordinary function definition (and that's essentially what it is, although types live in their own space where they can't be confused with functions of the same name).

The definition of a type thus consists of one or more type rules which basically have the same format as the general rules, but with the keyword `type` in front of each rule. Also, each left-hand side must have at most one argument pattern and exactly one right-hand side. Hence, if the definition of a type requires several right-hand sides, you normally have to write a separate `type` rule for each of them. Multiple left-hand sides work the same as in the general rule format, though.

As already mentioned, there is an alternative way for defining types in an indirect way through so-called interface types from which the corresponding type rules are derived automatically. These are part of Pure's declaration syntax and thus will be discussed later in the Declarations section. In this section we focus on how you can write your own type rules in order to define types in a direct fashion.

The identifier in the head of the left-hand side of a type rule is the name of the type which can then be used as a type tag in other equations, cf. Type Tags. This is just a normal, possibly qualified identifier subject to the same namespace mechanisms as other symbols; see Namespaces for details. However, as the type symbol only gets used as a type tag, it can never collide with function and variable symbols and hence the same symbol can be used both as a type and as a function or variable name.

A collection of type rules specifies a predicate, i.e. a unary, truth-valued function which denotes a set of terms. The type consists precisely of those terms for which the type predicate yields a nonzero result. For instance, the following type defines the type `triple` as the set of all tuples with exactly three elements:

```
type triple (x,y,z) = ~tuplep z;
```

Note that the type check consists of two parts here: The left-hand side pattern `(x,y,z)` restricts the set to all tuples with at least three elements. The right-hand side `~tuplep z` then verifies that the last component `z` is not a tuple itself, and thus the entire tuple consists of exactly three elements.

Another important point here is that the definition of the `triple` predicate is *partial*, as the given rule only applies to tuples with at least three elements. A value will only match the `triple` type tag if the predicate explicitly returns `true`; otherwise the match will fail, no matter what the result is (and even if the predicates just fails, i.e., returns an unevaluated normal form). Thus there is no need to make the predicate work on all terms (and in fact there are good reasons to *not* do so, see below).

In general, you should try to make your type definitions as specific as possible. This makes it possible to extend the predicate later, just like Pure allows you to extend the definition of a function to new types of arguments. For instance, if you later decide that lists with three elements should be considered as triples, too, then you may add the following type rule:

```
type triple [x,y,z] = true;
```

This makes it possible to define a type in a piecemeal fashion. Each subsequent rule enlarges the term set of the type. Conversely, consider a definition like:

```
type pair x = tuplep x && #x==2;
```

In this case the type rule applies to all values `x` and thus the type definition is complete; there is no way to extend it later. Whether to prefer the former or latter kind of definition depends on the situation. If you want to keep a type extensible, so that you can later make existing definitions of operations on the type work with new data representations, then you should use the former approach, otherwise the latter.

As an example for an extensible type definition, consider the following type `nat` which denotes the type of positive (machine) integers:

```
type nat x::int = x>0;
```

This definition is complete for the case of machine integers, but allows the type to be extended for other base types, and we'll do that in a moment. But first let's define the factorial on `nat` values as follows:

```
fact n::nat = if n==1 then 1 else n * fact (n-1);
```

Note that this definition would loop on zero or negative values if we permitted arbitrary `int` arguments. But since we restricted the argument type to `nat`, this case cannot occur and so the definition is safe:

```
> fact 0;
fact 0
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

The way we defined `fact`, it works on positive machine integers, but nothing else:

```
> fact 10L;
fact 10L
```

If we later decide that positive bigints should be considered as members of `nat` as well, we can simply add another rule for the `nat` type:

```
type nat x::bigint = x>0;
```

Et voila, our `fact` routine now magically works with bigints, too:

```
> map fact (0L..10L);
[fact 0L,1,2L,6L,24L,120L,720L,5040L,40320L,362880L,3628800L]
```

Note that we did all this without ever touching our original definition of `fact`. This works because the `bigint` data type already provides all the operations which we expect to use with the `nat` type. Pulling off this trick with other, more exotic kinds of data requires more preparation, since we'll first have to provide the required operations. In this case, we need at least multiplication, as well as comparisons with 1 and subtraction by 1. For instance, and just for the fun of it, let's implement our own variation of the `nat` type using Peano arithmetic:

```
type nat (s x) = true;

// addition
x + 0   = x;
x + 1   = s x;
x + s y = s (x+y);

// multiplication
x * 0   = 0;
x * 1   = x;
x * s y = x + x*y;

// subtract 1
s x - 1 = x;

// comparison with 0 and 1
s x == 0 = false;
s x == 1 = x == 0;
```

This implements just the bare bones, but that should be enough to make `fact` work. Let's give it a try:

```
> fact (s (s (s 0)));
s (s (s (s (s (s 0)))))
```

So, counting the `s`'s, the factorial of 3 is 6. Works! (It goes without saying, though, that this implementation of `nat` is not very practical; you'll get mountains of `s`'s for larger values of `n`.)

As you can see, a type definition may in general consist of many type rules which may be scattered out over different parts of a program. This works in exactly the same way as with ordinary functions.

There's an additional convenience provided for type rules, namely that the right-hand side may be omitted if it's just `true`. For instance, the rule

```
type nat (s x) = true;
```

from above can also be written simply as:

```
type nat (s x);
```

This kind of notation is particularly convenient for "algebraic types" which are usually given by a collection of constructors with different arities. For instance, a binary tree data type might be defined as follows (here we employ the | symbol to separate the different left-hand sides so that we can give all the constructor patterns in one go):

```
nonfix nil;
type bintree nil | bintree (bin x left right);
```

This method is also useful if you define your own abstract data types. In this case you're free to choose any suitable representation, so you might just wrap up all data objects of the type with a special constructor symbol, which makes checking the type simple and efficient. This is also the approach taken in the `point` example in Type Tags above, as well as by the container data types in the standard library.

The same notation can also be used to quickly make one type a "subtype" of another, or to create a type which is the union of several existing types. The following example can be found in the standard library:

```
type integer x::int | integer x::bigint;
```

A type rule can also take the form of a function definition without arguments. The corresponding right-hand side may either be another type symbol, or any kind of closure denoting a (curried) type predicate. In this case the defined type is simply an **alias** for the type denoted on the right-hand side. This is often done, e.g., for numeric types, to document that they actually stand for special kinds of quantities:

```
type speed = double;
type size = int;
```

Note that the definition of a type alias is always complete; there's no way to extend the corresponding type later. Therefore type aliases are normally resolved at compile time, so that they incur no additional runtime cost. For instance:

```
> half x::speed = x/2;
> show half
half x::double = x/2;
```

(If necessary, this "type folding" can also be disabled with the *`--nofold`* pragma.)

Finally, it's also possible to just specify the type name, without giving the right-hand side:

```
type thing;
```

This doesn't have any effect other than just declaring the type symbol, so that it can be used as a type tag in subsequent definitions. You then still have to give a proper definition of the type later (either as an explicit predicate or an alias).

Type aliases can also be used to quickly turn an existing predicate into a "convenience" type which can be used as a tag on the left-hand side of equations. The prelude defines a number of these, see *Prelude Types*. For instance:

```
type closure = closurep;
```

Conversely, you can turn any type tag into an ordinary predicate which can be used on the right-hand side of other definitions. To these ends, the prelude provides the `typep` predicate which takes a type symbol and the value to be checked as arguments. For instance:

```
type odd x::int = x mod 2;
type even x::int = ~odd x;

odd x = typep odd x;
even x = typep even x;
```

With those definitions you get:

```
> map odd (0..10);
[0,1,0,1,0,1,0,1,0,1,0]
> map even (0..10);
[1,0,1,0,1,0,1,0,1,0,1]
```

There's one caveat here. As the type symbol passed to `typep` gets evaluated in normal code you have to be careful if the symbol is also defined as a parameterless function or a variable; in such a case you'll have to quote the symbol, as described in section The Quote. For instance, we might rewrite the above definitions as follows, giving "pointless" definitions of the `odd` and `even` predicates in terms of `typep`:

```
type odd x::int = x mod 2;
type even x::int = ~odd x;

odd = typep ('odd);
even = typep ('even);
```

Note that the quotes on `odd` and `even` are really needed here to prevent the predicate definitions from looping. If you need this a lot then you might define a little helper macro (cf. Macros) which quotes the type symbol in an automatic fashion:

```
def typep ty::symbol = typep ('ty);
```

(However, this gets in the way if you want to check for computed type symbols, that's why this macro isn't defined in the prelude.)

Pure places no a priori restrictions on the rules defining a data type (other than that they must either define a unary predicate or an alias for an existing data type). As far as Pure is concerned, types are just subsets of the universe of terms. Thus any type of relation between two data types is possible; they might be unrelated (disjoint) term sets, one may be a subset of another, or they might be related in some other way (some terms may be members of both types, while others aren't).

For instance, consider the types `nat` and `odd` from above. Both are subtypes of the `int` type

(assuming our original definition of `nat` as the positive `int` values), but neither is a subtype of the other. It's sometimes useful to define the "intersection type" of two such types, which can be done in a straightforward way using the logical conjunction of the two type predicates:

```
type nat x::int = x>0;
type odd x::int = x mod 2;
type odd_nat x  = typep nat x && typep odd x;
```

Similarly, a variation of the `integer` union type from above could be defined using logical disjunction (this employs the `intp` and `bigintp` predicates from the prelude):

```
type myinteger x = intp x || bigintp x;
```

(Note that this isn't quite the same as the previous definition, which uses explicit patterns in order to make the definition extensible.)

Since the right-hand side of a type definition may in general be any predicate, it is up to the programmer to ensure that the definition of a type is actually computable. In fact, you should strive for the best possible efficiency in type predicates. A type definition which has worse than O(1) complexity may well be a serious performance hog depending on the way in which it is used, see Recursive Types in the Caveats and Notes section for more information about this.

Finally, note that in general it may be hard or even impossible to predict exactly when the code of a type definition will be executed at runtime. Thus, as a general rule, a type definition should not rely on side effects such as doing I/O (except maybe for debugging purposes), modifying references or external data structures via C pointers, etc.

1.5 Examples

This section assumes that you've read the Pure Overview and Rule Syntax sections, so that you are familiar with the basic elements of the Pure language. We now bring the pieces together and show you how simple but typical problems can be solved using Pure. You might use this section as a mini-tutorial on the Pure language. As we haven't discussed the more advanced elements of the Pure language yet, the scope of this section is necessarily limited. But it should give you a pretty good idea of how Pure programs looks like. After working through these examples you should be able to write useful Pure programs and understand the more advanced features discussed in subsequent sections.

1.5.1 Hello, World

The notorious "hello world" program can be written in Pure as follows:

```
using system;
puts "Hello, world!";
```

This employs the `puts` function from Pure's system module (which is in fact just the `puts` function from the C library). If you put these lines into a script file, say, hello.pure, you can run the program from the command line as follows:

```
$ pure hello.pure
Hello, world!
```

You may notice a slight delay when executing the script, before the "`Hello, world!`" message appears. That's because the interpreter first has to compile the definitions in your script as well as the prelude and other imported modules before the `puts "Hello, world!"` expression can be evaluated. The startup times can be reduced (sometimes considerably) by compiling scripts to native executables, see Compiled Scripts below.

Passing Parameters

Sometimes you may want to pass parameters to a script from the command line. The easiest way to do that is to invoke the interpreter as `pure -x` followed by the script name and the parameters. The interpreter then makes the command line parameters available as a list of strings in the `argv` variable. For instance, here is a version of the "hello world" program which uses `printf` to print the line `Hello, foo!` where `foo` is whatever was specified as the first command line parameter:

```
using system;
printf "Hello, %s!\n" (argv!1);
```

This script is invoked as:

```
$ pure -x hello.pure foo
Hello, foo!
```

Of course, many real-world programs will require more elaborate processing of command line parameters, such as recognizing program options. We won't discuss this here, but you can have a look at the `getopt` module which provides that kind of functionality in a convenient package.

Executable Scripts

It is often convenient if you can turn a script into a standalone executable which can be invoked by just typing its name on the command line. There are several ways to do this.

First, on most systems you can invoke the Pure script through some kind of shell script or command file which contains the command to invoke the interpreter. The details of this depend on the operating system and type of shell that you use, however, so we won't go into this here.

Second, on Unix-like systems it is possible to make any script file executable like this:

```
$ chmod a+x hello.pure
```

However, we also have to tell the shell about the command interpreter which should be invoked to run the script. (Otherwise the shell itself may try to execute the script, which won't work because it's not a shell script.) As already mentioned in Overview of Operation, this is done by adding a special kind of comment, a "shebang", to the beginning of the script, so that it looks like:

```
#!/usr/local/bin/pure -x
using system;
puts "Hello, world!";
```

Note that the `-x` option is only needed if the script actually processes its command line parameters. But it's a good idea to include it anyway, so that any extra arguments aren't accidentally read and processed by the interpreter instead.

Also note that you *must* give the full path to the Pure interpreter in the shebang line. This path of course depends on where you installed Pure. The above shebang will work with an installation from source, unless you changed the installation prefix when configuring the source package. If you installed the interpreter from a binary package, the proper path will often be `/usr/bin/pure` instead. In any case, you can find out where the interpreter lives by typing the following command in the shell:

```
$ which pure
/usr/local/bin/pure
```

If you get anything else on your system then you'll have to fix the shebang accordingly. You should then be able to run the script as follows:

```
$ ./hello.pure
Hello, world!
```

Note: Many modern Unix-like systems provide the `/usr/bin/env` utility which can perform a search for the interpreter executable, so that you can use a shebang like:

```
#!/usr/bin/env pure
```

This will always work if the Pure interpreter is installed somewhere on the system `PATH`. However, due to incompatibilities in the ways shebang lines are parsed by the shell on different Unix systems, the following might *not* work as expected:

```
#!/usr/bin/env pure -x
```

In particular, on Linux this causes `env` to look for an executable named "`pure -x`", which won't work. So, while `env` may be more convenient for simple shebangs, we generally advise against using it because of these limitations.

Compiled Scripts

Last but not least, you can also turn a Pure script into an executable by "batch-compiling" it. This works on all supported systems (provided that you have the necessary LLVM tools and 3rd party compilers installed, see the *installation instructions* for details). The result is a real native executable which can then be run directly just like any other binary program on your system. To these ends, the interpreter is run with the `-c` option which tells it to run in batch compilation mode, and the `-o` option which specifies the desired name of the executable. For instance:

```
$ pure -c hello.pure -o hello
Hello, world!
$ ./hello
Hello, world!
```

You'll notice that the compilation command in the first line above *also* prints the `Hello, world!` message. This reveals a rather unusual aspect of Pure's batch compiler: it actually *executes* the script even during batch compilation. The reasons for this behaviour and potential uses are discussed in the Batch Compilation section. If you want to suppress the program output during batch compilation, you can rewrite the program as follows:

```
using system;
main = puts "Hello, world!";
compiling || main;
```

Note that here we turned the code to be executed into a separate `main` function. This isn't really necessary, but often convenient, since it allows us to run the code to be executed by just evaluating a single function. (Note that in contrast to C, the name `main` has no special significance in Pure; it's just a function like any other. We still have to include a call to this function at the end of our program so that it gets executed.)

The last line now reads `compiling || main` which is a shorthand for "if the `compiling` variable is nonzero then do nothing, otherwise evaluate the `main` function". In a batch compilation the interpreter sets this variable to a nonzero value so that the evaluation of `main` is skipped:

```
$ pure -c hello.pure -o hello
$ ./hello
Hello, world!
```

We should mention here that batch-compiled scripts have some limitations because the compiled executable runs under a trimmed-down runtime system. This disables some of the advanced compile time features which are only available when running a script with the interpreter or at batch-compilation time. However, this won't usually affect run-of-the-mill scripts like the one above. More information about this can be found in the Batch Compilation section.

1.5.2 Running the Interpreter

While Pure scripts can be run as standalone programs directly from the shell, most of the time you'll probably use the Pure interpreter in an interactive way. You then simply run it like this:

```
$ pure

 __ \  |   |  __| _ \    Pure 0.56 (x86_64-unknown-linux-gnu)
 |   | |   | |    __/    Copyright (c) 2008-2012 by Albert Graef
 .__/ \__,_|_|  \___|    (Type 'help' for help, 'help copying'
_|                       for license information.)

Loaded prelude from /usr/lib/pure/prelude.pure.

>
```

The interpreter prints its sign-on message and leaves you at its command prompt. (You can also try `pure --plain` for a less fancy sign-on, or `pure -q` to completely suppress the message.)

At this point you can just start typing definitions and expressions to be evaluated. For instance:

```
> fact n = if n<=0 then 1 else n*fact (n-1);
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Note that Pure is a free-format language, and thus definitions and expressions *must* be terminated with a semicolon, so that the interpreter knows when you're done entering each item. This probably needs getting used to, but it's convenient because you can easily type more than one expression on a single line, or split longer constructs across multiple lines:

```
> 6*7; 16.3805*5.0;
42
81.9025
> 16753418726345
> * 991726534256718265234;
16614809890429729930396098173389730L
```

If the interpreter appears to just eat away expressions without printing any results, then most likely you forgot to enter the terminating semicolon. In such a case you can just type the semicolon on a line by itself:

```
> 6*7
> ;
```

(This won't do any harm even if it's not needed, because an empty item is always valid input at Pure's toplevel.)

The interpreter also reports syntax errors if you mistype an expression:

```
> 16.3805*(5;
<stdin>, line 8: syntax error, unexpected ';', expecting when or with or ')'
```

In such a case, just correct the error and resubmit the offending input. The interpreter's readline facility makes this pretty convenient, because you can use the cursor keys to recall previous input lines and edit them as needed.

Other kinds of errors may happen at runtime, when evaluating a syntactically correct expression. These give rise to so-called exceptions. For instance:

```
> 1 div 0;
<stdin>, line 9: unhandled exception 'signal 8' while evaluating '1 div 0'
```

Besides integer division by zero (flagged as '`signal 8`' here), common sources of exceptions are failed matches and conditionals, interrupts (e.g., if the user aborts an evaluation with `Ctrl-c`) and stack overflows (cf. Stack Size and Tail Recursion). Normally these are fatal and require you to fix the program or the expression that you entered, but programs can also catch these errors and handle them in any desired way, cf. Exception Handling.

Note that in contrast to most other programming languages, undefined identifiers are generally *not* an error in Pure. Instead, you'll simply get an unevaluated normal form:

```
> foo 5;
foo 5
```

Therefore, we recommend invoking the interpreter with the `-w` option so that it at least warns you about unknown symbols. You can also enter this option interactively or in a script using the `--warn` pragma:

```
> #! --warn
> bar 5;
<stdin>, line 12: warning: implicit declaration of 'bar'
bar 5
```

The interpreter has a global variable environment in which you can store intermediate results:

```
> let x = 16.3805*5;
> x; x/2; 1/x;
81.9025
40.95125
0.0122096395103935
> let y = 2*x; y;
163.805
```

Another handy feature is the special built-in function `ans` which yields the most recent result printed by the interpreter:

```
> 16.3805*5;
81.9025
> ans*2;
163.805
```

The interpreter recognizes a few other special commands which, like `ans`, are only available when it is run interactively. For instance, you can purge the value of a variable like this (this also works with any other defined item, such as constants, functions and macros):

```
> clear x
> x;
x
```

Another useful command is `show` which prints the definition of anything that you can define in a Pure script, such as variables and functions. For instance:

```
> show fact
fact n = if n<=0 then 1 else n*fact (n-1);
```

You can also just type `show` to print all definitions done interactively at the command prompt, which lets us review our accomplishments so far:

```
> show
fact n = if n<=0 then 1 else n*fact (n-1);
let y = 163.805;
```

The `dump` command saves these definitions in a file for later use:

```
> dump
```

This command doesn't print anything, but you can have a look at the written file in a text editor and maybe edit it as needed. By default, `dump` saves interactive definitions in a hidden file named `.pure` in the current directory, which gets reloaded automatically if we later rerun the interpreter in the same directory. We can also print this file, e.g., with the Unix `cat` command (note that '`!`' executes a shell command):

```
> !cat .pure
// dump written Wed Sep  5 10:00:15 2012
fact n = if n<=0 then 1 else n*fact (n-1);
let y = 163.805;
```

If we mess up badly, it's often convenient to just rerun the interpreter from scratch so that we can try again in a clean environment:

```
> run
```

As we've saved our scribblings with `dump` previously, those definitions will be reloaded automatically:

```
> show
fact n = if n<=0 then 1 else n*fact (n-1);
let y = 163.805;
```

If you don't want this then you can just remove the `.pure` file or rename it before invoking `run`.

Another helpful command is `help` which brings up the online documentation (this requires that you've configured the interpreter for the web browser that you use; see Online Help):

```
> help help
```

Last but not least, you can use the following command to exit the interpreter and return to the command shell:

```
> quit
```

Typing just an end-of-file character (usually `Ctrl-d` on Unix-like systems) at the beginning of the command line does the same.

There are a few other built-in commands that you may find useful when working with the interpreter, and you can even define your own. These interactive commands are special; they have their own syntax and need to be typed on a separate line. Please refer to Interactive Usage for a detailed explanation of the command syntax and the available commands.

1.5.3 Basic Examples

Pure has a few built-in data types, namely numbers (machine integers, bigints and double precision floating point numbers), strings, matrices, symbols, functions and pointer values. Compound expressions are formed from these using function application. In the syntax of the Pure language, these are also known as simple expressions. For want of a catchier name, we also simply call them **terms**. Pure is a programming language based on **term rewriting**, so all computations performed in Pure consist of the rewriting of terms. Some terms may reduce to other terms, others simply stand for themselves; the latter are also called **normal forms** and are what constitutes a "value" in the Pure language.

When the Pure interpreter starts up, it normally loads a collection of Pure scripts collectively called the **prelude**. The prelude defines many of the usual operations on numbers, strings, lists and other basic data structures that you may need, so you can start using the interpreter as a sophisticated kind of desktop calculator right away. Let's begin with some simple calculations involving integer and floating point numbers:

```
> 6*7;
42
> 16.3805*5.0;
81.9025
> 16753418726345 * 991726534256718265234;
16614809890429729930396098173389730L
```

Note that the integer constants in the last example exceeded the 32 bit range of machine integers, so they were promoted to bigints. The result is again a bigint (indicated by the `L` suffix). You can also turn *any* integer constant into a bigint by explicitly adding the `L` suffix:

```
> 6L*7L;
42L
```

Arithmetic with mixed operands will generally return the most general type capable of holding the result:

```
> 6*7L;
42L
> 16.3805*5;
81.9025
> 16.3805*5L;
81.9025
```

But note that most operations involving only machine integers will produce another machine integer; the result is *never* promoted to a bigint automatically, even in case of "overflow" (i.e., wrap-around). So the following will yield the same kind of signed 32 bit result as you'd get in C:

```
> 2147483647 + 1;
-2147483648
```

This has the advantage that you always know the type of the result of each operation beforehand by just looking at the types of the operands. It also makes it possible to compile machine integer operations to efficient native code. Therefore, if you suspect that a machine integer operation may wrap around and you'd thus prefer to do the calculation with bigints instead, you'll have to convert at least one of the operands to a bigint beforehand:

```
> 2147483647L + 1;
2147483648L
```

Also note that, in contrast to C or Fortran, the result of the `/` (division) and `^` (exponentiation) operators is *always* a floating point value in Pure, even if both operands are integers:

```
> 14/12;
1.16666666666667
> 2L^60L;
1.15292150460685e+18
```

Integer division and modulo are done with the `div` and `mod` operators, and exact powers of machine integers and bigints can be computed with the `pow` function:

```
> 14 div 12; 14 mod 12;
1
2
> pow 2 60;
1152921504606846976L
```

Also note that many of the standard math functions are available in a separate `math` module, so we need to import that module if we want to use one of these (see Modules and Imports for a detailed explanation of Pure's module system). For instance:

```
> using math;
> sqrt (16.3805*5)/.05;
181.0
```

The `math` module also provides you with complex and rational number types for doing more advanced calculations, but we won't go into that here.

Before we proceed, a few remarks about the syntax of function applications are in order.

Function application is an explicit operation in Pure, so that functions become first class values which can be passed around as function arguments and results. Like in most modern functional languages, function application is simply denoted by juxtaposition:

```
> sqrt 2;
1.4142135623731
```

In this case, you may also write `sqrt(2)` instead, but multiple arguments are normally specified as `f x y z` rather than `f(x,y,z)`. The former notation is known as **currying** (named after the American mathematician and logician Haskell B. Curry), and is ubiquitous in modern functional programming languages. The latter notation can be used in Pure as well, but it actually indicates that `f` is called on a *single*, structured argument (in this case a tuple). However, most predefined functions use the curried notation in Pure. For instance, the `max` function defined in the prelude takes two separate arguments, so it is invoked as follows:

```
> max 4 7;
7
```

Function application associates to the left, so the above is parsed as `(max 4) 7`, where `max 4` is called a **partial application** of the `max` function. A partial application is a function in its own right; e.g., `max 4` denotes the function which computes `max 4 y` for each given `y`.

Parentheses are used for grouping expressions as usual. In particular, since function application associates to the left, a nested function application in a function argument must be parenthesized as follows:

```
> sqrt (sqrt 2);
1.18920711500272
```

The same is true for any kind of expression involving operators, since function application binds stronger than any of these:

```
> sqrt (2*3);
2.44948974278318
```

The `map` function lets us apply a function to each member of a given list, which gives us a quick way of tabulating function values:

```
> map sqrt (0..2);
[0.0,1.0,1.4142135623731]
```

Here, the list argument is specified as an **arithmetic sequence** `0..2` which evaluates to the list `[0,1,2]`. This is fairly convenient when tabulating values of numeric functions. Here is another example which employs a partial application of the `max` function as the function argument:

```
> map (max 0) (-3..3);
[0,0,0,0,1,2,3]
```

Note that when the `max 0` function gets applied, say, to the first list member `-3`, we obtain the application `max 0 (-3)` which now has all the arguments that it needs; we also say that `max 0 (-3)` is a **saturated** application, which means that it's "ready to go". Evaluating `max 0`

`(-3)` gives `0` which becomes the first member of the result list returned by `map`. The other list members are calculated in an analogous fashion. It is easy to see that `max 0` thus computes what mathematicians call the "positive part" of its argument `x`, which is `x` itself if it is greater than `0` and `0` otherwise.

Operators aren't special either, they are just functions in disguise. You can turn any operator into an ordinary function by enclosing it in parentheses. Thus `(+)` denotes the function which adds its two arguments, and `x+1` can also be written as `(+) x 1`; in fact, the former expression is nothing but syntactic sugar for the latter. You can easily verify this in the interpreter:

```
> (+) x 1;
x+1
```

You can also have partial applications of operators like `(*) 2` which denotes a function which doubles its argument:

```
> map ((*) 2) [1,2,3,4,5];
[2,4,6,8,10]
```

Moreover, Pure offers some convenient syntactic sugar to denote so-called **operator sections** which specify a binary operator with only either its left or right operand. So the doubling function above may also be denoted as `(2*)` or `(*2)`. Similarly, `(+1)` denotes the "increment by 1" and `(1/)` the reciprocal function:

```
> map (+1) (1..5);
[2,3,4,5,6]
> map (1/) (1..5);
[1.0,0.5,0.333333333333333,0.25,0.2]
```

Note that the latter kind of section (also called a **left section**) is just a convenient shorthand for a partial application:

```
> (1/);
(/) 1
```

The former kind (a **right section**) can't be handled this way, because it's the *first* operand which is missing, and partial applications only allow you to omit trailing arguments. Instead, right sections expand to a partial application of the `flip` function,

```
> (+1);
flip (+) 1
```

which is defined in the prelude as follows:

```
flip f x y = f y x;
```

Note that `flip (+) 1` thus denotes a function which, when the missing operand is supplied, reduces to an application of the first (function) argument while also flipping around the operands. For another example, here's how you can compute third powers `3^x` of some numbers `x` with a right section of the '`^`' operator:

```
> map (^3) (1..5);
[1.0,8.0,27.0,64.0,125.0]
```

Note that this is exactly the same as:

```
> map (flip (^) 3) (1..5);
[1.0,8.0,27.0,64.0,125.0]
```

Such explicit applications of `flip` also work with ordinary functions like `pow`, so if we want to compute the cubes as exact bigint numbers, we can also write:

```
> map (flip pow 3) (1..5);
[1L,8L,27L,64L,125L]
```

Note the difference between `flip pow 3` which computes third powers, and `pow 3` which is a partial application that computes powers of 3.

Sometimes it is convenient to have function application as an explicit operation which can be passed as a function value to other functions. The `$` operator is provided for this purpose. `f $ x` is just `f x`, so you can write, e.g.:

```
> map ($1) [(+2),(*2),(/2)];
[3,2,0.5]
```

Recall that `($1)` is a right section which, when applied to an argument `f`, evaluates to `f $ 1 = f 1`. E.g., we have that `($1) (+2) = (+2) $ 1 = (+2) 1 = 2+1 = 3`. Hence the above expression actually applies a list of functions to the given argument `1`.

The `$` operator has a low precedence and is right-associative, so that it is sometimes used to eliminate the parentheses in cascading function calls. For instance, `sqrt $ sqrt $ 2*3` is the same as `sqrt (sqrt (2*3))`.

Another convenient operation for combining functions is the function composition operator, denoted '`.`'. It applies two functions in sequence, so that `(f.g) x` evaluates to `f (g x)`. For instance:

```
> g x = 2*x-1;
> map g (-3..3);
[-7,-5,-3,-1,1,3,5]
> map (max 0 . g) (-3..3);
[0,0,0,0,1,3,5]
```

Operations like '`.`', which take functions as arguments and return other functions as results, are also called **higher-order functions**. We'll have a closer look at these later.

As already mentioned, the interpreter also has a global variable environment in which you can store arbitrary expression values. This provides a means to define abbreviations for frequently-used expressions and for storing intermediate results. Global variable definitions are done with `let`. For instance:

```
> let x = 16.3805*5;
> x;
81.9025
```

As we've explained above, functions are first-class citizens and can thus be assigned to variables as well:

```
> let f = sqrt;
> f x/0.05;
181.0
```

The value of a global variable can be changed at any time. So we can type:

```
> let f = sin;
> f x/0.05;
4.38588407225469
```

You can also bind several variables at once by using an expression **pattern** as the left-hand side of a variable definition. This is useful if we need to extract elements from an aggregate value such as a list:

```
> let x1:x2:xs = map (^3) (1..5);
> x1,x2,xs;
1.0,8.0,[27.0,64.0,125.0]
```

Pure also provides a kind of "read-only" variables a.k.a. **constants**. They are defined pretty much like global variables (using the `const` keyword in lieu of `let`), but work more like a parameterless function whose value is precomputed at compile time:

```
> const π = 4*atan 1.0;
> show π
const π = 3.14159265358979;
> h x = sin (2*π*x);
> show h
h x = sin (6.28318530717959*x);
> map h [-1/4,-1/8,0,1/8,1/4];
[-1.0,-0.707106781186547,0.0,0.707106781186547,1.0]
```

Note that the compiler normally computes constant subexpressions at compile time, such as `2*π` in the function `h`. This works with all simple scalars (machine ints and doubles), see Constant Definitions for details.

As an aside, the last example also shows that Pure has no problems with Unicode. π is a Greek letter and thus an identifier as good as any other, although you will have a hard time finding that letter on an English keyboard. Fortunately, most operating systems nowadays provide you with an applet that lets you enter foreign language characters and other special symbols with ease.

1.5.4 Defining Functions

Now that we've learned how to run the interpreter and evaluate some expressions, it's time to embark on some real programming. Like in other functional programming languages, we do this by defining **functions** which perform the desired computation. The form these definitions take in Pure is a collection of **rewriting rules** which specify how an application

of the function reduces to another expression which then gets evaluated recursively to give the value of the function application.

In the simplest case, the left-hand side of a rewriting rule may just specify the function name along with some argument names. For instance:

```
square x = x*x;
```

Now, if we evaluate an expression like `square 7`, it reduces to `7*7` which in turn reduces to `49` by the built-in rules for integer arithmetic. You can verify this by entering the definition in the interpreter:

```
> square x = x*x;
> square 7;
49
```

In fact, the above definition is completely generic; since `x` is an unqualified variable, we can apply `square` to *any* value `x` and have it evaluate to `x*x`:

```
> square 7.0;
49.0
> square 7L;
49L
> square (a+b);
(a+b)*(a+b)
```

As the last example shows, this will even work if the supplied argument is no number at all, which is useful, e.g., if we want to do symbolic evaluations.

Functions can have as many arguments as you like, subject to the constraint that each equation defining the function has the *same* number of arguments on the left-hand side. For instance, suppose that we want to calculate the sum of two squares. We can do this using the `square` function from above as follows:

```
> sumsquares x y = square x + square y;
> sumsquares 3 4;
25
```

The interpreter keeps track of the number of arguments of each defined function, so if we accidentally try to define `sumsquares` with three arguments later then we'll get an error message:

```
> sumsquares x y z = square x + square y + square z;
<stdin>, line 8: function 'sumsquares' was previously defined with 2 args
```

This actually makes perfect sense if you think about the way curried function applications work. If the above was permitted, then an expression like `sumsquares x y` would become ambiguous (would it denote an invocation of the binary `sumsquares` or a partial application of the ternary one?).

Thus Pure doesn't really have **variadic** functions which take a variable number of arguments. There are ways to emulate this behaviour in some cases, but usually it's easier to just pass the arguments as a single structured value instead. It is customary to employ tuples

for this purpose, so that the call uses the familiar notation `f (x,y,z)`. A typical example are optional arguments. For instance, suppose that we'd like to define a function `incr` which increments a numeric value, where the amount to be added can be specified as an optional second value which defaults to 1. This can be done in Pure as follows:

```
incr (x,y) = x+y;
incr x = x+1 otherwise;
```

These equations *must* be in the indicated order. Pure considers different equations for the same function in the order in which they are written. Therefore "special case" rules, like the one for `incr (x,y)` in this example, must be listed first. (Note that if the second equation came first, `incr (5,2)` would reduce to `(5,2)+1` rather than `5+2`, because `x` also matches, in particular, any tuple `x,y`.)

Functions taking a single tuple argument are also (somewhat misleadingly) called **uncurried** functions, because their arguments have to be given all in one go, which precludes partial applications of the function. While curried functions are often preferred, uncurried functions can be more convenient at times, e.g., if you have to map a function to a list containing given combinations of arguments. For instance, given the above definition of `incr` we may write:

```
> map incr [(5,1),(5,2),(6,3),(7,5)];
[6,7,9,12]
```

To make this work with curried functions, the prelude provides a function `uncurry` which turns a curried function of two arguments into an uncurried one which takes a single tuple argument:

```
> map (uncurry (+)) [(5,1),(5,2),(6,3),(7,5)];
[6,7,9,12]
```

On the other hand, some generic list processing functions such as `foldl` expect curried functions, so the reverse transformation `curry` is also provided:

```
> foldl (curry incr) 0 (1..10);
55
```

In fact, the definitions of `curry` and `uncurry` don't involve any special magic, they just translate curried calls to uncurried ones and vice versa. From the horse's mouth:

```
> show curry uncurry
curry f x y = f (x,y);
uncurry f (x,y) = f x y;
```

A function can also have zero arguments, i.e., you can define parameterless functions such as:

```
foo = 1..3;
```

The function is then simply invoked without any arguments:

```
> foo;
[1,2,3]
```

It is worth noting the difference between this and the variable definition:

```
let bar = 1..3;
```

While `bar` and `foo` yield the same result `[1,2,3]`, they do so in different ways. `bar` is a global variable whose value is computed once and then stored under its name, so that the value can be simply recalled when `bar` is later invoked in an expression. Also, the value of `bar` can be changed at any time with an appropriate `let` statement. (If the value is not supposed to change later then you can also define it as a `const` instead.)

In contrast, `foo` is a function which recomputes the list value on each invocation. To avoid the overhead of recalculating the same value each time it is needed, a variable or constant is usually preferred over a parameterless function in Pure. However, a parameterless function will be needed if the computation involves some hidden side effects which cause a new value to be produced for each invocation. For instance, the `math` module provides a parameterless function `random` which computes a new pseudo random number each time it is called:

```
> using math;
> random, random, random;
-795755684,581869302,-404620562
```

Many functions also involve conditionals which let them take different computation paths depending on the outcome of a condition. One way to do this is to employ a **conditional expression**. For instance, we may compute the sign of a number as follows:

```
> sign x = if x>0 then 1 else if x<0 then -1 else 0;
> map sign (-3..3);
[-1,-1,-1,0,1,1,1]
```

Alternatively, you can also use a collection of **conditional rules** instead:

```
sign x =  1 if x>0;
       = -1 if x<0;
       =  0 otherwise;
```

Note that here we omitted the left-hand side in the second and third equations, in which case the compiler assumes that it's the same as for the first equation; cf. Rule Syntax for details. Also note that the `otherwise` keyword is only syntactic sugar in Pure, you can always omit it. However, it tends to improve readability by marking the default case of a definition.

Both styles are frequently used in Pure programs; it depends on the situation which one is more appropriate. Conditional rules make the conditions stick out more clearly and hence tend to improve readability. On the other hand, conditional expressions can be nested more easily and thus facilitate the programming of complicated decision trees.

Function definitions may also be recursive, i.e., a function may invoke itself either directly or indirectly in its definition. For instance, here is a definition of the Ackerman function using conditional rules:

```
ack x y = y+1 if x == 0;
        = ack (x-1) 1 if y == 0;
        = ack (x-1) (ack x (y-1)) otherwise;
```

We will have more to say about recursive functions later; see Recursion below.

1.5.5 Pattern Matching

So far we have only seen function definitions involving just unqualified variables as parameters. In general it is possible to specify arbitrary patterns for the parameters, in which case the actual arguments are checked against the patterns and, if everything matches up, the right-hand side of the rule is invoked with the variables in the patterns bound to their corresponding values.

The simplest nontrivial patterns are type tags which can be placed on a variable to restrict the type of value an argument can match. For instance:

```
> square x::int = x*x;
> square 7;
49
```

Note that in contrast to our previous generic definition of the `square` function we gave in Defining Functions, this definition now only applies to the case of an `int` argument:

```
> square 7.0;
square 7.0
```

Polymorphic definitions can be made by giving separate equations for the different argument types. For instance, we can easily add an equation for the `double` case:

```
> square x::double = x*x;
> show square
square x::int = x*x;
square x::double = x*x;
> square 7; square 7.0;
49
49.0
```

Here the right-hand sides of both rules are the same. Pure has a convenient shorthand notation for this case which lets you factor out the common right-hand side using the '`|`' delimiter as follows:

```
square x::int | square x::double = x*x;
```

The compiler expands this to the same two rules as above:

```
square x::int = x*x;
square x::double = x*x;
```

Let's compare this to our earlier generic definition of `square`:

```
square x = x*x;
```

There are two different kinds of polymorphism at work here. The latter, generic definition is an example of **parametric polymorphism**; it applies to *any* type of argument `x` whatsoever (at least if it makes sense to multiply a member of the type with itself). Also note that

this definition is "closed"; because equations are considered in the order in which they are written, there's no way you could add another "special case" rule to this definition later.

In contrast, the former definition leaves any application of `square` to a value other than `int` or `double` undefined. This gives us the opportunity to define `square` on as many types of arguments as we like, and (this is the crucial point) define the function in *different* ways for different argument types. This is also known as **ad-hoc polymorphism** or **function overloading**. For instance, if we later need to square 2x2 matrices, we might add a rule like:

```
square {a,b;c,d} = {a*a+b*c,a*b+b*d;c*a+d*c,c*b+d*d};
```

Pure places no restriction on the number of equations used to define a function, and the different equations may in fact be scattered out over many different places. So as long as the left-hand side patterns properly discriminate between the different cases, you can overload any operation in Pure to handle as many argument types as you want. However, it is important to note that in contrast to overloaded functions in statically typed languages such as C++, there's really only *one* `square` function here which handles all the different argument types. The necessary "dispatching" to select the proper rewriting rule for the argument values at hand is done at runtime by pattern matching.

Parametric polymorphism has the advantage that it lets you define polymorphic functions in a very concise way. On the other hand, ad-hoc polymorphism lets you deal with disparate cases of an operation which cannot easily be reconciled. It also allows you to tailor the definition to the specific case at hand, which might be more efficient than using a generic rule. You can also combine both approaches, but in this case you have to list the special case rules before the generic ones. For instance:

```
square x::int | square x::double |
square x = x*x;
```

(Note that the first two rules are just specialization of the last rule to `int` and `double` arguments, so we could in fact eliminate the special case rules here and still get the same results. But the type tags tell the compiler that the argument in these rules is always an `int` or `double`, respectively, so it may generate more efficient code for these cases.)

Patterns may also involve constant values, in which case the constant must be matched literally in the argument. For instance, here is another definition of the Ackerman function from Defining Functions which uses constant argument patterns instead of conditional rules:

```
ack 0 y = y+1;
ack x 0 = ack (x-1) 1;
ack x y = ack (x-1) (ack x (y-1)) otherwise;
```

The first two rules take care of the "base cases" `x==0` and `y==0`. Note that these rules *must* be given in the indicated order to make them work. Specifically, the left-hand side `ack x y` of the last equation also matches, in particular, terms like `ack 0 y` and `ack x 0`, so placing the last equation before the first two will "shadow" those rules and cause non-termination, resulting in a stack overflow. Similarly, placing the second equation before the first one will cause the definition to loop on `ack 0 0`.

Another point that deserves mentioning here is that constants on the left-hand side of a

rule *must* be matched literally, cf. Constant Patterns. E.g., `ack 0 y` only matches if the first argument is really `0`, not `0.0` or `0L` (although these compare equal to `0`). So the above definition of `ack` isn't quite the same as our previous definition from Defining Functions. If you wanted the definition above to also work with double and bigint values, you'd have to add corresponding rules for the `0.0` and `0L` cases.

Last but not least, patterns are also used to "deconstruct" structured values like lists, tuples and matrices, binding variables to the component values. For instance, to compute the sum of a list of values, you may write:

```
> sum [] = 0;
> sum (x:xs) = x+sum xs;
> sum (1..100);
5050
```

This definition works in a straightforward recursive manner. The first rule involves the constant pattern `[]` and thus handles the base case of an empty list, in which case the sum is zero. The second rule has a structured argument pattern `x:xs` which denotes a list with head element `x` and tail `xs`; in this case the result is `x` added to the sum of the remaining list elements `xs`. (In fact, this computational pattern is so common that the prelude provides a family of functions such as `foldl` and `foldr` to do this kind of operation in a generic way. Our `sum` function above is actually equivalent to `foldr (+) 0`, see List Processing below for details.)

Instead of placing the patterns directly into the left-hand sides of the function definition, you might also do the necessary pattern-matching in the right hand side, by employing a `case` expression:

```
sum xs = case xs of [] = 0; x:xs = x+sum xs end;
```

This works a bit different, though, since a `case` expression raises an exception if the target expression is not matched (cf. Patterns):

```
> sum (1:2:xs);
<stdin>, line 2: unhandled exception 'failed_match' while evaluating 'sum (1:2:xs)'
```

To avoid that, you may want to add a type tag, which ensures that the argument of `sum` is of the proper type:

```
sum xs::list = case xs of [] = 0; x:xs = x+sum xs end;
```

Now the case of an improper list is handled a bit more gracefully, yielding the same normal form expression you'd get with the first definition of `sum` above:

```
> sum (1:2:xs);
1+(2+sum xs)
```

Pure also allows to define `sum` in a more traditional way which will be familiar to Lisp programmers (note that `head` and `tail` correspond to Lisp's "car" and "cdr"):

```
sum xs::list = if null xs then 0 else head xs + sum (tail xs);
```

Choosing one or the other is again a question of style. However, if you're dealing with concrete data structures such as lists, pattern-matching definitions are often more convenient and easier to understand.

Pattern matching also works with user-defined constructors (cf. Data Types). For instance, here's how to implement an insertion operation which can be used to construct a binary tree data structure useful for sorting and searching:

```
nonfix nil;

insert nil y         = bin y nil nil;
insert (bin x L R) y = bin x (insert L y) R if y<x;
                     = bin x L (insert R y) otherwise;
```

Note that `nil` needs to be declared as a `nonfix` symbol here, so that the compiler doesn't mistake it for a variable; see The "Head = Function" Rule for details. The following example illustrates how the above definition may be used to obtain a binary tree data structure from a list:

```
> tree [] = nil;
> tree (x:xs) = insert (tree xs) x;
> tree [7,12,9,5];
bin 5 nil (bin 9 (bin 7 nil nil) (bin 12 nil nil))
```

Conversely, it's also easy to convert such a tree structure back to a list. We can then combine these operations to sort a list in ascending order:

```
> list nil = [];
> list (bin x L R) = list L + (x:list R);
> list (tree [7,12,9,5]);
[5,7,9,12]
```

1.5.6 Local Functions and Variables

Up to this point our examples only involved global functions and variables. When the problems to be solved become more difficult, it will be necessary to structure the solution in some way, so that you'll often end up with many small functions which need to work in concert to solve the problem at hand. Typically only a few of these functions will serve as actual entry points, while other functions are only to be used internally. Pure supports this through **local** functions and variables whose scope is limited either to the right-hand side of a rule or one of its subexpression. This offers two main advantages:

- Local functions and variables are hidden from the main scope so that they can only be used in the context where they are needed and don't clutter up the global environment. This provides a way to define functions in a modular fashion while hiding internal details from the rest of the program.
- The right-hand sides of local definitions have full access to other local functions and variables in their parent environments, which eliminates the "plumbing" which would otherwise be needed to pass these values around. For instance, a local function nested

in another function can freely access the parent function's arguments and other local variables in its scope.

Local functions are defined using the `with` construct, while local variables can be introduced with a `when` or `case` expression, see Special Expressions for details. These constructs can be tacked on to any expression, and they can also be nested. For instance:

```
> f 5 with f x = y+y when y = x*x end end;
50
```

Note that the local function `f` there computes twice the square of its argument `x`. To these ends, first `x*x` is assigned to the local variable `y` whose value is then doubled by computing `y+y` which becomes the result of `f`.

Local functions can also be created without actually naming them, by employing a so-called **lambda abstraction**. For instance, a function which squares its argument might be denoted as `\x -> x*x`. This is pretty much the same as a local function `f with f x = x*x end` except that the function remains nameless. This notation is pretty convenient for making up little "one-off" functions which are to be applied on the spot or passed as function arguments or results to other functions. For instance, here's how you can compute the first ten squares, first with an ordinary (named) local function, and then with an equivalent lambda:

```
> map f (1..10) with f x = x*x end;
[1,4,9,16,25,36,49,64,81,100]
> map (\x -> x*x) (1..10);
[1,4,9,16,25,36,49,64,81,100]
```

For obvious reasons lambdas work best for non-recursive functions. While there are techniques to create recursive functions out of lambdas using so-called fixed point combinators (cf. `fix`), named functions are much more convenient for that purpose.

Pattern matching works in local definitions as usual. For instance, here are several ways to swap two values represented as a tuple, using either a local function or a `when` or `case` expression:

```
> swap (1,2) with swap (x,y) = y,x end;
2,1
> (\(x,y) -> y,x) (1,2);
2,1
> y,x when x,y = 1,2 end;
2,1
> case 1,2 of x,y = y,x end;
2,1
```

You'll also frequently find code like the following, where a global "wrapper" function just sets up some initial parameter values and then invokes a local "worker" function which does all the real work. The following function calculates the sum of the positive integers up to `n` (the "accumulating parameters" technique used in this example will be explained later, cf. Recursion).

```
sum n = sum 0 n with
  sum s n = s if n < 0;
          = sum (s+n) (n-1) otherwise;
end;
```

Note that there are actually *two* separate functions named `sum` here. This works because according to the scoping rules the right-hand side of the global definition is under the scope of the `with` clause, and thus the call `sum 0 n` on the right-hand refers to the *local* `sum` function, not the global one. (While it is perfectly correct and even makes sense in this example, this style may be somewhat confusing, so we often prefer to give wrapper and worker different names for clarity.)

As discussed in Scoping Rules, a local function can refer to other local functions and variables in its parent environments. It can also be returned as a function value, which is where things get really interesting. The local function value then becomes a **lexical closure** which carries around with it the local variable environment it was created in. For instance:

```
> adder x = add with add y = x+y end;
> let g = adder 5; g; map g (1..5);
add
[6,7,8,9,10]
```

Note that here the local function `add` refers to the argument value `x` of its parent function `adder`. The invocation `adder 5` thus returns an instance of `add` which has `x` bound to the value `5`, so that `add y` reduces to `5+y` for each `y`. This works as if this instance of the `add` closure had an invisible `x` argument of `5` attached to it. (And this is in fact how closures are implemented internally, using a transformation called lambda lifting which effectively turns local functions into global ones.) You should study this example carefully until you fully understand how it works; we'll see a bunch of other, more complicated examples of this kind later.

Lexical closures also provide a means to encapsulate data in a way reminiscent of object-oriented programming. For instance:

```
nonfix coords;

point (x,y) = \msg -> case msg of
  coords = x,y;
  move (dx,dy) = point (x+dx,y+dy);
end;
```

The anonymous function returned by `point` in fact works like an "object" which can be queried for its coordinates and moved by a given offset through corresponding "messages" passed as arguments to the object:

```
> let p = point (1,2); p;
#<closure 0x7f420660e658>
> p coords; p (move (2,3)) coords;
1,2
3,5
```

Note that this still lacks some typical features of object-oriented programming such as mutability and inheritance. It isn't really hard to add these, but this requires the use of some of Pure's more advanced machinery which we didn't discuss yet. For instance, mutability can be implemented in Pure by using so-called *expression references*, a kind of mutable storage cells which can hold arbitrary expression values:

```
> let x = ref 99; get x;
99
> put x 2;
2
> get x;
2
```

Using these we can rewrite our definition of the `point` object as follows:

```
nonfix coords;

point (x,y) = (\msg -> case msg of
  coords = get x, get y;
  move (dx,dy) = put x (get x+dx), put y (get y+dy);
end) when
  x,y = ref x,ref y;
end;
```

Note that the coordinates are kept in corresponding expression references assigned to the local `x` and `y` variables, which now shadow the `x` and `y` arguments of `point`. This makes it possible to have `move` actually modify the `point` object in-place:

```
> let p = point (1,2); p coords;
1,2
> p (move (2,3)); p coords;
3,5
3,5
```

It goes without saying that this style isn't preferred in functional programs, but it certainly has its uses, especially when interfacing to imperative code written in other languages such as C.

1.5.7 Data Types

Before we consider the more advanced uses of functions in Pure, a few remarks about data types are in order. Like Lisp, Pure is basically a "typeless" language. That doesn't mean that there are no data types; in fact, they're a dime a dozen in Pure. But Pure lets you make up your own data structures as you go, without even formally defining a data type. Data types *can* be defined and associated with a name pretty much in the same way as functions, but that's just a convenience and completely optional. This sets Pure apart from statically typed languages like ML and Haskell, where explicit data type definitions are mandatory if you want to introduce new data structures.

As we've seen, Pure knows about a few built-in types such as numbers, strings, symbols

and functions; everything else is a function application. If a symbol is defined as a function, which merely means that there are some rewriting rules for it, then an application of that function to some arguments may evaluate to something else. But if it doesn't, then Pure is perfectly happy with that; it just means that the function application is in normal form and thus becomes a "value". For instance:

```
> cons 3 (cons 5 nil);
cons 3 (cons 5 nil)
```

There's nothing mysterious about this; the `cons` and `nil` symbols being used here aren't defined anywhere, and thus any terms constructed with these symbols are just "data", no questions asked. We also call such symbols **constructors**. (Note that these are different from constructors in object-oriented programming; constructor applications in term rewriting and functional programming normally don't execute any code, they're just literal data objects.)

We can now go ahead and define some operations on this kind of data. (To these ends, it's necessary to declare `nil` as a nonfix symbol so that we can use it as a literal in patterns; cf. Pattern Matching.)

```
nonfix nil;

#nil = 0;
#cons x xs = #xs+1;

head (cons x xs) = x;
tail (cons x xs) = xs;

nil + ys = ys;
cons x xs + ys = cons x (xs + ys);
```

Et voilà, we've just created our own list data structure! It's admittedly still a bit paltry, but if we keep at it and define all the other functions that we need then we could turn it into a full-blown replacement for Pure's list data structure. In fact Pure's lists work in a very similar fashion, using the infix `':'` constructor and the empty list `[]` in lieu of `cons` and `nil`, respectively.

If we want, we can define a new data type for the data structure we just invented. This works by giving a number of type rules similar to those used in function definitions. In general, these may denote arbitrary unary predicates, but in our case it's sufficient to just list the patterns of terms which are supposed to be members of the type (see Type Rules for an explanation of the definition syntax):

```
type mylist nil | mylist (cons x xs);
```

This definition lets us use the `mylist` type as a tag on the left-hand side of an equation, cf. Pattern Matching. But if we're content with using the patterns directly then we might just as well do without that.

Types consisting solely of constructor term patterns are sometimes also called **algebraic types**. In fact, most user-defined data structures are algebraic types in Pure, and there are plenty of examples of these in the standard library as well. In particular, lists and tuples

are algebraic types, as are complex and rational numbers, and most of Pure's container data types such as dictionaries and sets are also implemented as algebraic types.

Pure differs from most functional languages in that symbols may act as *both* constructors and defined functions, depending on the arguments. Thus Pure allows you to have "constructors with equations". For instance:

```
cons nil ys = ys;
cons (cons x xs) ys = cons x (cons xs ys);
```

Now `cons` has become a (partially) defined function. Note that these rules make `cons` associative and turn `nil` into a left-neutral element for `cons`. This in fact makes `cons` behave like concatenation, so that our lists are always flat now:

```
> cons (cons 1 (cons 2 nil)) (cons 3 nil);
cons 1 (cons 2 (cons 3 nil))
```

Examples of such constructor equations can be found in the standard library as well, such as the rules used to flatten tuples, keep rational numbers in lowest terms, or confine the angles of complex numbers in polar notation.

Another possible use of constructor equations is to check the well-formedness of constructor terms. For instance, in our example we might want to preclude terms like `cons 1 2` which don't have a `mylist` in the second argument to `cons`. This can be done with a constructor equation which raises an exception in such cases (cf. Exception Handling):

```
> cons x y = throw (bad_mylist y) if ~typep mylist y;
> cons 1 2;
<stdin>, line 18: unhandled exception 'bad_mylist 2' while evaluating 'cons 1 2'
```

A specific kind of algebraic data types which are useful in many applications are the **enumerated types**. In this case the type consists of symbolic constants (nonfix symbols) only, which are the elements of the type. For instance:

```
nonfix sun mon tue wed thu fri sat;
type day sun | day mon | day tue | day wed | day thu | day fri | day sat;
```

However, to make this type actually work as an enumerated type, we may want to provide definitions for basic arithmetic, `ord`, `succ`, etc. This is rather straightforward, but tedious. So as of Pure 0.56, the standard library provides a little utility module, `enum`, which generates the necessary definitions in an automatic fashion. All we have to do is to import the module and then invoke the `enum` function on the type and we're set:

```
using enum;
enum day;
```

It's also possible to define the type and make it enumerable in one go using the `defenum` function:

```
defenum day [sun,mon,tue,wed,thu,fri,sat];
```

In either case, we can now perform calculations with the members of the type just like with other predefined enumerated types such as numbers and characters:

```
> ord sun;
0
> day (ans+3);
wed
> pred sat;
fri
> sun+3;
wed
> fri-2;
wed
> fri-tue;
3
> mon..fri;
[mon,tue,wed,thu,fri]
> sun:tue..sat;
[sun,tue,thu,sat]
> sat:fri..mon;
[sat,fri,thu,wed,tue,mon]
```

A more abstract way to define algebraic types are the interface types. For instance, if we take another look at the operations defined on our list type, we may observe that the data structure is quite apparent from the patterns in the rules of operations such as '#' and '+'. Pure lets us leverage that information by creating an algebraic type from a collection of operation patterns it supports. For instance, we may write:

```
interface list_alike with
  #x::list_alike;
  x::list_alike + y;
end;
```

This defines a generic type consisting of all terms which may be passed as an argument to both '#' and '+'. We can ask the interpreter about the patterns actually matched by the type as follows:

```
> show interface list_alike
type list_alike s::string;
type list_alike [];
type list_alike (x:xs);
type list_alike nil;
type list_alike (cons x xs);
```

Note that the `list_alike` type not only includes our own list type, but also any other data structure providing the '#' and '+' operations. This also comprises the standard list and string types for which there are definitions of the '#' and '+' operations in the prelude.

Pure's interface types are a first attempt at formalizing the notion of Duck typing in Pure. They are thus still a bit experimental and require some diligence in defining the interface operations in a suitable way. Please check Interface Types in the Declarations section for more information and examples.

1.5.8 Recursion

Recursion means that a function calls itself, either directly or indirectly. It is one of the most fundamental techniques in functional programming, and you won't find many useful Pure programs which don't use it in one form or another. That's because most interesting programs execute pieces of code repeatedly. Pure doesn't have any special looping constructs, so recursion is the only way to do this in Pure. We've already seen various examples of this throughout the manual, so let's take a closer look at it now and learn a few related tricks along the way.

For a simple example, consider the factorial. In order to compute the factorial of an integer `n`, we need to multiply the positive integers up to `n`. There's a straightforward recursive definition which does this:

```
fact n = if n>0 then n*fact (n-1) else 1;
```

If you prefer conditional rules instead, you can also write:

```
fact n = n*fact (n-1) if n>0;
       = 1 otherwise;
```

It's not hard to see how this definition operates. The first rule only applies if `n>0`, otherwise the second rule kicks in so that `fact n` becomes `1` if `n` is zero or negative (which is consistent with our informal description because in this case the product of all positive integers up to `n` is the empty product which is `1` by mathematical convention). The first rule is the interesting one where the recursion happens. If `n>0` then we may compute `fact n` by computing `fact (n-1)` recursively and multiplying that with `n`, giving `n*(n-1)*...*1`. Let's check that this works:

```
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Note that these numbers grow fairly quickly; they outgrow the 32 bit range and start wrapping around already at `n==13`. To avoid that, you'll have to do the computation with bigints, or you could use floating point values if you don't mind the limited precision.

```
> fact 13;
1932053504
> fact 13L;
6227020800L
> fact 30L;
265252859812191058636308480000000L
> fact 30.0;
2.65252859812191e+32
```

However, you'll run into another, more serious obstacle if you want to compute factorials for some really big values of `n`. That's because each recursive invocation of `fact` needs some small amount of memory on the execution stack, a so-called "stack frame". (If you're not familiar with how subroutine calls are executed by keeping the execution context on a stack then you should revisit your basic computer science lessons now.) When `n` becomes

big enough then the stack space may be exhausted, which usually results in a fatal runtime error:

```
> fact 1000000L;
Segmentation fault
```

The Pure interpreter can optionally do stack checks so that at least you get an orderly exception when this happens, see Stack Size and Tail Recursion for details. To enable this, you'll need to set the PURE_STACK environment variable to the maximum stack size in kilobytes. E.g., on Unix-like systems you can type the following in a Bourne-compatible shell, or put this command into your shell's startup file:

```
$ export PURE_STACK=1024
```

(Note that most systems will provide much more stack space than 1024 kilobytes. We deliberately picked a small limit here so that the stack overflows are detected sooner.)

Now start the interpreter again, enter the above definition of `fact` and try it with some large value of `n`. You'll get something like:

```
> fact 100000L;
<stdin>, line 3: unhandled exception 'stack_fault' while evaluating 'fact 100000L'
```

Ok, so we now get an orderly exception, which is much better than being kicked out of the interpreter. But this doesn't really solve the problem. How can we avoid using all that stack space in the first place? In a language like C we'd be using a specialized loop construct instead of recursion. But this isn't an option in Pure. Instead, we can rewrite the definition so that it becomes **tail-recursive**. This means that the recursive call becomes the final operation on the right-hand side of the recursive rule.

The trick of the trade to turn a recursive function into a tail-recursive one is the **accumulating parameter** technique. The idea here is to have a separate "worker" function which carries around an extra argument representing the intermediate result for the current iteration. The final value of that parameter is then returned as the result. In the case of the factorial this can be done as follows:

```
fact n = loop n 1 with
  loop n p = loop (n-1) (n*p) if n>0;
           = p otherwise;
end;
```

Note that `fact` has now become a simple "wrapper" which supplies the initial value of the accumulating parameter (`p` in this case) for the "worker" function `loop` which does all the hard work. This kind of design is fairly common in functional programs.

Our worker function is tail-recursive since the recursive call to `loop` is indeed the final call on the right-hand side of the first equation defining `loop`. The Pure compiler generates code which optimizes such "tail calls" so that they reuse the stack frame of the calling function. Thus a tail-recursive function like `loop` will execute in constant stack space. And in fact after entering our new definition of `fact` we can now compute `fact 100000L` just fine:

```
> fact 100000L;
2824229407960347874293421578024535518477... // lots of digits follow
```

The accumulating parameter technique isn't fully general, but it covers all the kinds of simple iterative algorithms which you'd do using loop constructs in traditional programming languages. Some algorithms may require additional techniques such as **tabulation** (keeping track of some or all intermediate results), however, so that they can be written in an iterative form. To see how this can be done in Pure, let's consider the Fibonacci numbers. These can be computed with the following naive recursive definition:

```
fib n = if n<=1 then n else fib (n-2) + fib (n-1);
```

Here are some members of this famous sequence:

```
> map fib (0..20);
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

Note that the right-hand side of the definition above involves *two* recursive invocations of `fib` in the `else` branch. This is bad because it means our definition will need exponential running time. (More precisely, you'll find that the ratio between the running times of successive invocations quickly starts approaching the golden ratio $\phi = 1.618...$, which is no accident because the times are proportional to the Fibonacci function itself!)

Using a simple iterative algorithm, it is possible to calculate the Fibonacci numbers in linear time instead. Observe that each member of the sequence is simply the sum of the two preceding members. If we keep track of the last two members of the sequence then we can compute the next member with a single addition. This yields the following tail-recursive implementation which uses the same kind of "wrapper-worker" design:

```
fib n = loop n 0L 1L with
  loop n a b = loop (n-1) b (a+b) if n>0;
             = a otherwise;
end;
```

Note that as a matter of prudence we primed the iteration with the bigints `0L` and `1L` so that we can compute large Fibonacci numbers without suffering wrap-around. For instance, try the following. (Sit back and relax, this takes a little while; the result has 208988 digits.)

```
> fib 1000000;
19532821287077577316320149475962563324 43... // lots of digits follow
```

Recursion also naturally occurs when traversing recursive data structures. We've already seen various examples of these, such as the binary tree data structure:

```
nonfix nil;

insert nil y         = bin y nil nil;
insert (bin x L R) y = bin x (insert L y) R if y<x;
                     = bin x L (insert R y) otherwise;
```

The `insert` function implements a binary tree insertion algorithm which keeps the tree (rep-

resented with the `bin` and `nil` constructor symbols) sorted. To these ends, it recurses into the left or right subtree, depending on whether the element `y` to be inserted is less than the current element `x` or not. The final result is a new tree which has a `nil` subtree replaced with a new `bin y nil nil` subtree at the right location.

If we do an **inorder** traversal of such a binary tree (at each non-`nil` subtree, first visit the left subtree, then note the element at the top of the current subtree, and finally visit the right subtree), we obtain the elements of the tree in ascending order. This traversal is also implemented recursively, e.g., as follows:

```
list nil = [];
list (bin x L R) = list L + (x:list R);
```

Note that these functions can't be made tail-recursive using the accumulating parameter technique, because traversing a tree structure requires more general forms of recursion. There is in fact a more general continuation passing technique to do this, which we will look at in The Same-Fringe Problem below; alas, it's not as easy as accumulating parameters. Fortunately, some important recursive structures such as lists only involve simple recursion and can thus be traversed and manipulated in a tail-recursive fashion more easily. For instance, consider our earlier definition of the `sum` function:

```
sum [] = 0;
sum (x:xs) = x+sum xs;
```

This definition isn't tail-recursive, but we can easily massage it into this form using the accumulating parameter technique:

```
sum xs::list = loop 0 xs with
  loop s [] = s;
  loop s (x:xs) = loop (s+x) xs;
end;
```

Functions can also be **mutually recursive**, in which case two or more functions are defined in terms of each other. For instance, suppose that we'd like to skip every other element of a list (i.e., return a list with only the elements having either even or odd indices, respectively). One way to do this involves two functions (named `pick` and `skip` here) which recursively call each other:

```
> pick [] = []; pick (x:xs) = x:skip xs;
> skip [] = []; skip (x:xs) = pick xs;
> pick (1..10);
[1,3,5,7,9]
> skip (1..10);
[2,4,6,8,10]
```

A Numeric Root Finder

Let's now see how we can apply the techniques explained above in the context of a somewhat more practical example: a numeric root finder. That is, we're going to write a function

which takes another function `f` and determines a (double) value `x` such that `f x` becomes (close to) zero.

We'll develop this in a bottom-up fashion. The method we employ here is known as the Newton-Raphson algorithm, whose basic building block is the following routine `improve` which improves a given candidate solution `x` by computing a first-order approximation of the root. This involves computing (a numeric approximation of) the first derivative at the given point, which we do using a second function `derive`:

```
improve f x = x - f x / derive f x;
derive f x = (f (x+dx) - f x) / dx;
```

If you still remember your calculus then these should look familiar. Note that in both functions, `f` is our target function to be solved and `x` the current candidate solution. The second equation is nothing but the difference quotient of the function at the point `x`, using `dx` as the increment along the `x` axis. The `improve` function computes the intersection of the corresponding secant of `f` with the `x` axis.

To illustrate how the method works, let's perform a few improvement steps manually, using the target function `f x = x*x-2` which becomes zero at the square root of 2. Here we choose a `dx` value of `1e-8` and start from the initial guess `2`:

```
> let dx = 1e-8;
> improve f x = x - f x / derive f x;
> derive f x = (f (x+dx) - f x) / dx;
> f x = x*x-2;
> improve f 2;
1.49999999696126
> improve f ans;
1.41666666616021
> improve f ans;
1.41421568628522
> improve f ans;
1.41421356237468
```

It should be apparent by now that this converges to the square root of 2 rather quickly. To automate this process, we need another little helper function which iterates `improve` until the current candidate solution is "good enough". A suitable termination criterion is that the improvement drops below a certain threshold (i.e., `abs (x-f x) <= dy` for some reasonably small `dy`). For extra safety, we'll also bail out of the loop if a prescribed number `n` of iterations has been performed. This function can be implemented in a tail-recursive fashion as follows:

```
loop n f x = x if n <= 0;
  = if abs (x-y) < dy then y else loop (n-1) f y when y = f x end;
```

Let's give it a try:

```
> let dy = 1e-12;
> loop n f x = x if n <= 0;
>   = if abs (x-y) < dy then y else loop (n-1) f y when y = f x end;
> loop 20 (improve dx f) 2;
1.4142135623731
```

```
> ans*ans;
2.0
```

Looks good. So let's finally wrap this up in a main entry point `solve` which takes the function to be solved and an initial guess as parameters. Our little helper functions `improve`, `derive` and `loop` are only used internally, so we can turn them into local functions of `solve`. The additional parameters of the algorithm are implemented as global variables so that we can easily modify their values if needed. The end result looks as follows. Note that the initial guess `x` is an implicit parameter of the `solve` function, so the function actually gets invoked as `solve f x`.

```
let dx = 1e-8;  // delta value for the approximation of the derivative
let dy = 1e-12; // delta value for testing convergence
let nmax = 20;  // maximum number of iterations

solve f = loop nmax (improve f) with
  loop n f x = x if n <= 0;
    = if abs (x-y) < dy then y else loop (n-1) f y when y = f x end;
  improve f x = x - f x / derive f x;
  derive f x = (f (x+dx) - f x) / dx;
end;
```

Here are some examples showing how the `solve` function is used. Note that we specify the target functions to be solved as lambdas here. E.g., `\t -> t^3-x` denotes a function mapping `t` to `t^3-x`, which becomes zero if `t` equals the cube root of `x`.

```
> sqrt x = solve (\t -> t*t-x) x;
> sqrt 2; sqrt 5;
1.4142135623731
2.23606797749979
> cubrt x = solve (\t -> t^3-x) x;
> cubrt 8;
2.0
```

Our little root finder isn't perfect. It needs a fairly well-behaved target function and/or a good initial guess to work properly. For instance, consider:

```
> solve (\t -> 1/t-2) 1;
0.00205230175365927
```

Here `solve` didn't find the real root at 0.5 at all. In fact, if you print the solution candidates then you will find that `solve` converges rather slowly in this case and thus bails out after 20 iterations before a good solution is found. Increasing the `nmax` value fixes this:

```
> let nmax = 50;
> solve (\t -> 1/t-2) 1;
0.5
```

There are other pathological cases where the algorithm performs even more poorly. Further improvements of the method presented here can be found in textbooks on numeric algorithms; the interested reader may want to cut his teeth on these algorithms by translating them to Pure in the way we've shown here.

The Same-Fringe Problem

This is one of the classical problems in functional programming which has a straightforward recursive solution, but needs some thought if we want to solve it in an efficient way. Consider a (rooted, directed) tree consisting of branches and leaves. To keep things simple, we may represent these structures as nested lists, e.g.:

```
let t1 = [[a,b],c,[[d]],e,[f,[[g,h]]]];
let t2 = [a,b,c,[[d],[],e],[f,[g,[h]]]];
let t3 = [[a,b],d,[[c]],e,[f,[[g,h]]]];
```

Thus each inner node of the tree is represented as a list containing its (zero or more) subtrees, and the leaves are the "atomic" (non-list) elements. The **fringe** of such a structure is the list of all leaves in left-to-right order, which can be computed as follows:

```
fringe t = if listp t then catmap fringe t else [t];
```

Note that `listp` is a predicate which decides whether its argument is a (proper or improper) list and the `catmap` function applies the given function to a list, like `map`, and concatenates all the resulting lists, like `cat`. Thus, if the argument `t` is an "atom" (leaf) then `fringe` simply returns `[t]`, otherwise it recursively applies itself to all the subtrees and concatenates the results:

```
> fringe t1;
[a,b,c,d,e,f,g,h]
> fringe t2;
[a,b,c,d,e,f,g,h]
> fringe t3;
[a,b,d,c,e,f,g,h]
```

Note that `t1` and `t2` differ in structure but have the same fringe, while `t1` and `t3` have the same structure but different fringes. The problem now is to decide, given any two trees, whether they have the same fringe. Of course, we can easily solve this by just computing the fringes and comparing them with '`===`' (note that we employ syntactic equality here which also allows us to compare symbols, for which '`==`' isn't normally defined):

```
> fringe t1 === fringe t2;
1
> fringe t3 === fringe t2;
0
```

However, this is rather inefficient since we always have to fully construct the fringes which may need considerable extra time and space if the trees are large. Most of this effort may be completely wasted if we only need to inspect a tiny fraction of the fringes to find out that they're different, as in the case of `t2` and `t3`. Also note that our version of the `fringe` function isn't tail-recursive and we may thus run into stack overflows for large trees.

This problem, while posed in an abstract way here, is not only of academic interest. For instance, trees may be used as an alternative string data structure which implements concatenation in constant time by just delaying it. In this case we certainly don't want to explicitly carry out all those concatenations in order to decide whether two such objects are the same.

Therefore, this problem has been studied extensively and more efficient approaches have been developed. One way to solve the problem involves the technique of **continuation passing** which is a generalization of the accumulating parameter technique we already discussed. It never constructs any part of the fringes explicitly and also works in constant stack space. The algorithm can be implemented in Pure as follows. (This is a slightly modified transliteration of a Lisp program given in Henry Baker's article "Iterators: Signs of Weakness in Object-Oriented Languages", ACM OOPS Messenger 4(3), 1993, pp. 18-25, which is also available from Henry Baker's Archive of Research Papers.)

```
samefringe t1 t2 =
samefringe (\c -> genfringe t1 c done) (\c -> genfringe t2 c done) with
  done c = c [] done;
  samefringe g1 g2 =
    g1 (\x1 g1 -> g2 (\x2 g2 -> x1===x2 && (x1===[] || samefringe g1 g2)));
  genfringe [] c g = g c;
  genfringe (x:t) c g = genfringe x c (\c -> genfringe t c g);
  genfringe x c g = c x g;
end;
```

As Baker admits himself, this style of programming isn't "particularly perspicuous", so we'll explain the algorithm in a moment. But first let us verify that the program indeed works as advertized. It's helpful to print out the actual comparisons performed in the innermost lambda in the definition of the local `samefringe` function, which can be done by adding a little debugging statement as follows (this also needs an import clause "`using system;`" to make the `printf` function available):

```
samefringe g1 g2 =
  g1 (\x1 g1 -> g2 (\x2 g2 -> printf "%s === %s?\n" (str x1,str x2) $$
                              x1===x2 && (x1===[] || samefringe g1 g2)));
```

With this we get:

```
> samefringe t1 t2;
a === a?
b === b?
c === c?
d === d?
e === e?
f === f?
g === g?
h === h?
[] === []?
1
```

So in this case we do a complete traversal of both trees which is the best that we can hope for if the fringes are the same. Note that the final comparison `[] === []` ensures that we also hit the end of the two fringes at the same time. This test deals with the corner case that one fringe is a prefix of the other. For instance:

```
> let t4 = [[a,b],c,[[d]],e,[f,[[g,h,i]]]];
> samefringe t4 t2;
a === a?
```

```
b === b?
c === c?
d === d?
e === e?
f === f?
g === g?
h === h?
i === []?
0
```

Things go a bit differently, however, when comparing `t3` and `t2`; as soon as we hit the first discrepany between the two fringes, the algorithm bails out and correctly asserts that the fringes are different:

```
> samefringe t3 t2;
a === a?
b === b?
d === c?
0
```

Let's take a closer look at the various parts of the algorithm now. First, the `genfringe` function:

```
genfringe [] c g = g c;
genfringe (x:t) c g = genfringe x c (\c -> genfringe t c g);
genfringe x c g = c x g;
```

This routine generates the fringe of a tree, given as the first argument, on the fly. The second argument `c` (the "consumer") is a function which gets invoked on the current leaf, to do any required processing. (As we'll see later, it may also get invoked with the special "sentinel" value `[]` to indicate the end of the fringe.)

The third argument `g` (the "generator") is a **continuation**, a kind of "callback function" to be invoked *after* the current subtree has been traversed, in order to process the remainder of the tree. It takes the consumer function `c` as its sole argument. Consequently, `genfringe` simply invokes the continuation `g` on the consumer `c` when applied to an empty subtree `[]`, i.e., if there aren't any leaves to be processed. This case is handled in the first equation for `genfringe`.

The second equation for `genfringe` is the interesting one where the recursion happens. It deals with a nonempty tree `x:t` by invoking itself recursively on `x`, setting up a new continuation `\c -> genfringe t c g`, which will take care of processing the rest of the subtree `t`, after which it chains to the previous continuation `g` which will handle the rest of the tree.

The third equation for `genfringe` handles the case of a non-list argument, i.e., a leaf. In this case we just pass the leaf `x` to the consumer function `c` along with the continuation `g`. The consumer processes `x` as needed and may then decide to call the continuation `g` on itself in order to continue processing the rest of the tree, or simply bail out, returning any value. Note that this entire process is tail-recursive, as long as `c` chains to `g` as the last call. It thus only needs constant stack space in addition to what `c` itself uses.

Note that we need an initial continuation `g` to get the process started. This is provided by the `done` function:

```
done c = c [] done;
```

As we've defined it, `done` invokes the consumer `c` on an empty list to signal the end of the fringe. For good measure, it also passes itself as the continuation argument; however, normally the consumer will never use this argument and just bail out when invoked on the `[]` value.

To see how this works, we can just enter `done` and `genfringe` as global functions and invoke them on a suitable consumer function, e.g.:

```
> done c = c [] done;
> genfringe [] c g = g c;
> genfringe (x:t) c g = genfringe x c (\c -> genfringe t c g);
> genfringe x c g = c x g;
> c x g = if x===[] then g else printf "%s... " (str x) $$ g c;
> genfringe t1 c done;
a... b... c... d... e... f... g... h... done
```

In the case of `samefringe`, we use the local `samefringe` function as our consumer instead. This works pretty much the same, except that `samefringe` employs *two* continuations `g1` and `g2` to traverse both trees at the same time:

```
samefringe g1 g2 =
  g1 (\x1 g1 -> g2 (\x2 g2 -> x1===x2 && (x1===[] || samefringe g1 g2)));
```

Note that the outer lambda `(\x1 g1 -> ...)` becomes the consumer for the first generator `g1` which traverses `t1`. When called, it then invokes the second generator `g2`, which traverses `t2`, on the consumer (inner lambda) `(\x2 g2 -> ...)`. This in turn does the necessary tests to verify that the current leaf elements are the same, or to bail out from the recursion if they aren't or if we reached the end of the fringes. Also note that this is still tail-recursive because the short-circuit logical operations `&&` and `||` are both tail-recursive in their second operand (cf. Stack Size and Tail Recursion).

1.5.9 Higher-Order Functions

As we have seen, functions are first-class citizens in Pure which can be created on the fly (using partial applications as well as lambdas and local functions), assigned to variables and passed around freely as function arguments and results. Thus it becomes possible to define **higher-order functions** which take other functions as arguments and/or return them as results. This is generally considered a hallmark feature of functional programming, and much of the power of functional programming stems from it. In fact, higher-order functions are so deeply ingrained in the modern functional programming style that you'll hardly find a nontrivial program that doesn't use them in some way, and we have already seen many examples of them throughout the manual. While most imperative programming languages today let you treat functions as values, too, they're typically much more limited in the ways that new functions can be created dynamically. Only recently have partial application and

anonymous closures arrived in some mainstream imperative languages, and they are often still rather awkward to use.

The simplest case of a higher-order function is a function which takes another function as an argument. For instance, we have seen the function `map` which applies a function to each member of a list. If it wasn't in the prelude, it could be defined as follows:

```
map f [] = [];
map f (x:xs) = f x : map f xs;
```

(Note that this isn't the actual definition from the prelude, which goes to some lengths to make the operation tail-recursive and properly handle lazy lists. But we won't dive into these technicalities here since we're only interested in the higher-order aspect right now.)

This definition is rather straightforward: To map a function `f` to a list, just apply it to the head element `x` and recurse into the tail `xs`. The recursion stops at the empty list which is returned as is. For instance:

```
> map (*2) (0..10);
[0,2,4,6,8,10,12,14,16,18,20]
```

The prelude includes an entire collection of such generic list functions which have proven their utility as basic building blocks for many list processing tasks. We'll have a closer look at these later, see List Processing.

Another numerical example is the function `derive` which we used in our root finder example to calculate the difference quotient of a function `f` at a given point `x`:

```
derive f x = (f (x+dx) - f x) / dx;
```

This example is also interesting because we can turn `derive` into a function mapping functions to other functions, by partially applying it to the target function. So we may write:

```
> let dx = 1e-8;
> map (derive square) (1..4) with square x = x*x end;
[1.99999998784506,3.99999997569012,5.99999996353517,7.99999995138023]
```

This illustrates an easy way to create new functions from existing ones: partial application. (In fact we also did that when we applied the operator section `(*2)` using `map` above. Note that `(*2)` is a function which doubles its single argument.) This simple recipe is surprisingly powerful. For instance, the prelude defines the function composition operator '`.`' as:

```
(f.g) x = f (g x);
```

The partial application `f.g` thus applies two given functions `f` and `g` in sequence (first `g`, then `f`). Functions of this kind, which create new functions by combining existing ones, are also known as **combinators**. For instance, using '`.`' we can easily create a function which "clamps" its argument between given bounds by just combining the `min` and `max` functions from the prelude as follows:

```
> clamp a b = max a . min b;
> map (clamp (-3) 3) (-5..5);
[-3,-3,-3,-2,-1,0,1,2,3,3,3]
```

Note that partial application works with constructor symbols, too:

```
> map (0:) [1..3,4..6,7..9];
[[0,1,2,3],[0,4,5,6],[0,7,8,9]]
```

Another more direct way to define combinators is to make them return a local or anonymous function. For instance, the following equations lift the '+' and '-' operators to pointwise operations:

```
f + g = \x -> f x + g x if nargs f > 0 && nargs g > 0;
f - g = \x -> f x - g x if nargs f > 0 && nargs g > 0;
```

This employs the `nargs` function from the standard library which returns the argument count of a global or local function. We use this here to check that the operands are defined functions taking at least one argument. The result is a function which applies the function operands to the given argument and computes their sum and difference, respectively. For instance:

```
> map (f+g-h) (1..10) with f x = 2*x+1; g x = x*x; h x = 3 end;
[1,6,13,22,33,46,61,78,97,118]
```

These rules also handle functions taking multiple arguments, so that you can write, e.g.:

```
> (max-min) 2 5;
3
```

Constructors can be extended in exactly the same way:

```
> f,g = \x -> f x, g x if nargs f > 0 && nargs g > 0;
> (max,min,max-min) 2 5;
5,2,3
```

1.5.10 List Processing

Pure's list data structure provides you with a convenient way to represent sequences of arbitrary values. This is one of the few compound data structures which has built-in support by the compiler, so that some syntactic sugar is available which allows you to express certain list operations in a convenient way. But for the most part, lists are implemented in the prelude just like any other data structure.

The empty list is denoted `[]`, and compound lists can be put together in a right-recursive fashion using the ':' operator. The customary bracketed notation is provided as well, and this is also the syntax the interpreter normally uses to print list values:

```
> 1:2:3:[];
[1,2,3]
```

Note that the bracketed notation is just syntactic sugar; internally all list values are represented as right-recursive applications of the ':' operator. Thus it is possible to match the head and tail of a list using a pattern like `x:xs`:

```
> case [1,2,3] of x:xs = x,xs end;
1,[2,3]
```

Lists can contain any combination of elements (also from different types) and they may also be nested:

```
> [1,2.0,[x,y],"a string"];
[1,2.0,[x,y],"a string"]
```

List concatenation is denoted `+`, and the `#`, `!` and `!!` operators can be used to compute the length of a list and extract elements and slices of a list using zero-based indexing:

```
> [a,b,c]+[x,y,z];
[a,b,c,x,y,z]
> #ans, ans!5, ans!![2,3];
6,z,[c,x]
```

Note that lists are immutable in Pure (just like most of Pure's built-in and predefined data structures), so there are no operations which modify lists in-place. E.g., concatenation works as if it was defined recursively by the following rules:

```
[]+ys = ys;
(x:xs) + ys = x : (xs+ys);
```

So a new list is created which replaces the empty list in the last component of the left operand with the right operand. This even works if the second operand is no list at all, in which case an improper list value is produced:

```
> [a,b,c]+y;
a:b:c:y
```

These can be useful, e.g., to represent symbolic list values. Note that a **proper** list value contains the empty list `[]` in its rightmost component; an **improper** list value is one which doesn't. There are some list functions like `reverse` which really need proper lists to work and will throw an exception otherwise, but many predefined operations will deal with improper lists just fine:

```
> map f (x:y:z);
f x:f y:map f z
```

Lists can also be compared using the `==` and `~=` operators:

```
> [1,2,3] == [1,2,4];
0
```

Arithmetic sequences are denoted with the `..` operator:

```
> 1..10; 10:9..1; 0.0:0.1..1.0;
[1,2,3,4,5,6,7,8,9,10]
[10,9,8,7,6,5,4,3,2,1]
[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
```

List comprehensions provide another way to construct (proper) list values using a convenient math-like notation:

```
> [2^x | x = 1..10];
[2.0,4.0,8.0,16.0,32.0,64.0,128.0,256.0,512.0,1024.0]
```

We'll discuss this construct in more detail later, see List Comprehensions.

The prelude provides a fairly comprehensive collection of useful list functions, including some powerful generic operations which let you do most common list manipulations with ease. For instance, we have already seen the `map` function:

```
> map (\x->2*x-1) (1..10);
[1,3,5,7,9,11,13,15,17,19]
```

There's also a function `do` which works in the same fashion but throws away all the results and simply returns `()`. Of course this makes sense only if the applied function has some interesting side-effect. E.g., here's a quick way to print all members of a list, one per line. This combines the `str` function (which converts any Pure expression to its printable representation, cf. String Processing below) with the `puts` function from the `system` module (which is just the corresponding C function, so it prints a string on the terminal, followed by a newline).

```
> using system;
> do (puts.str) (1..3);
1
2
3
()
```

Another useful list function is `filter` which applies a predicate to each member of a list and collects all list elements which satisfy the predicate:

```
> odd x = x mod 2; even x = ~odd x;
> filter odd (1..20);
[1,3,5,7,9,11,13,15,17,19]
> filter even (1..20);
[2,4,6,8,10,12,14,16,18,20]
```

In addition, the `all` and `any` functions can be used to check whether all or any list elements satisfy a given predicate:

```
> any even (1:3..20);
0
> all odd (1:3..20);
1
```

There's also a family of functions such as `foldl` which generalize the notion of aggregate functions such as list sums and products. Starting from a given initial value `a`, `foldl` iterates a binary function `f` over a list `xs` and returns the accumulated result. It's defined as follows:

```
foldl f a [] = a;
foldl f a (x:xs) = foldl f (f a x) xs;
```

For instance, we can use `foldl` to compute list sums and products:

```
> foldl (+) 0 (1..10);
55
> foldl (*) 1 (1..10);
3628800
```

Note that `foldl` ("fold-left") accumulates results from left to right, so the result accumulated so far is passed as the *left* argument to the function `f`. There's a `foldr` ("fold-right") function which works analogously but collects results from right to left, and accordingly passes the accumulated result in the *right* argument. Usually this won't make a difference if the iterated function is associative, but `foldl` and `foldr` have lots of applications beyond these simple use cases. For instance, we may use `foldl` to reverse a list as follows:

```
> foldl (flip (:)) [] (1..10);
[10,9,8,7,6,5,4,3,2,1]
```

Note that we have to flip the arguments of the `':'` constructor here, since `foldl` passes the accumulated list in the left argument, but `':'` wants it on the right. Conversely, we have that:

```
> foldr (:) [] (1..10);
[1,2,3,4,5,6,7,8,9,10]
```

This just returns the list unchanged. So the order in which we accumulate results does matter here.

In a similar fashion, we might use `foldl` (or `foldr`) to build any kind of compound data structure from a list of its members. For instance, recall our binary tree example:

```
nonfix nil;
insert nil y         = bin y nil nil;
insert (bin x L R) y = bin x (insert L y) R if y<x;
                     = bin x L (insert R y) otherwise;
```

We can then use `foldl insert` to construct a binary tree from its member list as follows:

```
> foldl insert nil [7,12,9,5];
bin 7 (bin 5 nil nil) (bin 12 (bin 9 nil nil) nil)
```

Sometimes we'd like to know not just the final result of an aggregate function, but all the intermediate results as well. The `scanl` function does this. For instance:

```
> scanl (+) 0 (1..10);
[0,1,3,6,10,15,21,28,36,45,55]
```

Note that this computes the same list of partial sums as:

```
> [foldl (+) 0 (1..n) | n = 0..10];
[0,1,3,6,10,15,21,28,36,45,55]
```

However, the former is more efficient since it does all the partial sums in one go.

Like `foldl`, `scanl` also has a sibling called `scanr` which collects results from right to left, starting at the end of the list:

```
> scanr (+) 0 (1..10);
[55,54,52,49,45,40,34,27,19,10,0]
```

Another useful list generation function is `iterwhile` which keeps applying a function starting at a given initial value, as long as the current value satisfies the given predicate. So another way to generate the odd numbers up to 20 is:

```
> iterwhile (<=20) (+2) 1;
[1,3,5,7,9,11,13,15,17,19]
```

Or we might collect all powers of 2 which fall into the 16 bit range:

```
> iterwhile (<0x10000) (*2) 1;
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768]
```

There are also various functions to partition a list into different parts according to various criteria. The simplest of these are the `head` and `tail` functions:

```
> let xs = 1..10;
> head xs; tail xs;
1
[2,3,4,5,6,7,8,9,10]
```

Conversely, the `last` and `init` functions give you the last element of a list, and all but the last element, respectively:

```
> last xs; init xs;
10
[1,2,3,4,5,6,7,8,9]
```

The `take` and `drop` functions take or remove a given number of initial elements, while `takewhile` and `dropwhile` take or remove initial elements while a given predicate is satisfied:

```
> take 4 xs; drop 4 xs;
[1,2,3,4]
[5,6,7,8,9,10]
> takewhile (<=4) xs; dropwhile (<=4) xs;
[1,2,3,4]
[5,6,7,8,9,10]
```

Lists can be reversed with `reverse` and sorted using `sort`:

```
> reverse xs;
[10,9,8,7,6,5,4,3,2,1]
> sort (<) (xs + ans);
[1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
```

You can also concatenate a list of lists with the `cat` function:

```
> cat [1..n | n = 1..5];
[1,1,2,1,2,3,1,2,3,4,1,2,3,4,5]
```

Last but not least, there is the `zip` family of functions which let you combine members of two or more lists in different ways. The `zip` function itself collects pairs of corresponding elements in two input lists:

```
> zip (1..5) ("a".."e");
[(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")]
```

The effect of `zip` can be undone with `unzip` which returns a pair of lists:

```
> unzip ans;
[1,2,3,4,5],["a","b","c","d","e"]
```

The `zipwith` function is a generic version of `zip` which combines corresponding members from two lists using a given binary function `f`:

```
> zipwith (*) (1..10) (1..10);
[1,4,9,16,25,36,49,64,81,100]
```

You might also consider `zipwith` a variant of `map` working with two lists at the same time (in fact this operation is also known as `map2` in some functional programming languages). There are also variations of these functions which work with three lists (`zip3`, `unzip3`, `zipwith3`).

Note that `zip` itself is equivalent to `zipwith (,)`:

```
> zipwith (,) (1..5) ("a".."e");
[(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")]
```

Also note that since tuples are formed by just applying the ',' operator repeatedly, you can use multiple calls of `zip` to piece together tuples of any length:

```
> zip (1..3) (zip ("a".."c") [a,b,c]);
[(1,"a",a),(2,"b",b),(3,"c",c)]
```

This can be achieved even more easily by folding `zip` over a list of lists; here we employ a variation `foldr1` of `foldr` which takes the initial value from the beginning of the list.

```
> foldr1 zip [1..3,"a".."c",[a,b,c]];
[(1,"a",a),(2,"b",b),(3,"c",c)]
```

Note that this method easily scales up to as many element lists as you want. Recovering the original element lists is a bit trickier, though, but it can be done using this little helper function:

```
unzipn n xs = xs if n<=1;
            = xs,unzipn (n-1) ys when xs,ys = unzip xs end otherwise;
```

For instance:

```
> foldr1 zip [1..3,"a".."c",[a,b,c]];
[(1,"a",a),(2,"b",b),(3,"c",c)]
```

```
> unzipn 3 ans;
[1,2,3],["a","b","c"],[a,b,c]
```

Also, the elements to be zipped don't have to be singletons, they can themselves be tuples of any size:

```
> foldr1 zip [[1,2,3],[a,(),c],[x,y,(z,t)]];
[(1,a,x),(2,y),(3,c,z,t)]
```

But note that in this case you loose the information which elements came from which sublists, so `unzip` won't be able to recover the original lists any more. If you need to avoid that then it's best to use other aggregates such as lists or vectors for the sublist elements.

There are other interesting list functions in the prelude, but we'll leave it at that for now. Please check the *Pure Library Manual* for a full account of the available operations.

1.5.11 String Processing

Let's take a short break from lists and look at strings. We postponed that until now since strings are in many ways just like lists of characters. In fact the similarities run so deep that in some languages, most notably Haskell, strings *are* in fact just lists. Pure doesn't go quite that far; it still represents strings as null-terminated arrays of characters in the UTF-8 encoding, which is a much more compact representation and eases interoperability with C. However, most common list operations also work on strings in an analogous fashion. Thus you can concatenate strings, compute their length, and index, slice and compare them as usual:

```
> "abc"+"xyz";
"abcxyz"
> #ans, ans!5, ans!![2,3];
6,"z","cx"
> "abc"=="abd";
0
```

In addition, strings can also be ordered lexicographically:

```
> "abd"<"abcd";
0
> "abd">"abcd";
1
> sort (<) ["the","little","brown","fox"];
["brown","fox","little","the"]
```

Where it makes sense, list operations on strings return again a string result:

```
> head "abc"; tail "abc";
"a"
"bc"
> take 4 "abcdefg"; drop 4 "abcdefg";
"abcd"
"efg"
```

A slight complication arises with the map function, because in this case the result is not guaranteed to be a string in all cases. For instance:

```
> map ord "HAL";
[72,65,76]
```

To have map work consistently, it will thus yield a list even in cases where the result *could* again be represented as a string. If you want a string result instead, you'll have to do the conversion explicitly, using the string function:

```
> map (+1) "HAL";
["I","B","M"]
> string ans;
"IBM"
```

Conversely, you can also convert a string to a list of its characters using either chars or the generic list conversion function:

```
> list ans;
["I","B","M"]
```

As in the case of map, this conversion is usually done automatically if a list operation from the prelude is applied to a string. This also happens if a list comprehension draws values from a string:

```
> [x-1 | x = "IBM"];
["H","A","L"]
```

Talking about characters, these are simply single character strings, so Pure has no separate data type for them. However, there is a type tag char for the single character strings which can be used in pattern matching:

```
> isupper x::char = "A"<=x && x<= "Z";
> filter isupper "The Little Brown Fox";
"TLBF"
> any isupper "The Little Brown Fox";
1
```

Maybe you wondered how that "HAL" => "IBM" transformation above came about? Well, the prelude also defines basic arithmetic on characters:

```
> "a"+1, "a"+2, "z"-1;
"b","c","y"
> "z"-"a";
25
```

This considers characters as an enumerated data type where each character corresponds to a numeric code point in Unicode. Hence, e.g., "a"+1 gives "b" because "b" is the code point following "a" in Unicode, and "b"-"a" gives 1 for the same reason.

So here's the rot13 encoding in Pure:

```
rot13 x::string = string (map rot13 x) with
  rot13 c = c+13 if "a" <= lower c && lower c <= "m";
          = c-13 if "n" <= lower c && lower c <= "z";
          = c otherwise;
  lower c = "a"+(c-"A") if "A"<=c && c<="Z";
          = c otherwise;
end;
```

For instance:

```
> rot13 "The quick brown fox";
"Gur dhvpx oebja sbk"
> rot13 ans;
"The quick brown fox"
```

Character arithmetic also makes arithmetic sequences of characters work as expected:

```
> "a".."k"; "k":"j".."a";
["a","b","c","d","e","f","g","h","i","j","k"]
["k","j","i","h","g","f","e","d","c","b","a"]
> string ("a":"c"..."z");
"acegikmoqsuwy"
```

You can also convert between characters and their ordinal numbers using the `ord` and `chr` functions:

```
> ord "a";
97
> chr (ans+1);
"b"
```

Thus using Horner's rule we might convert a string of decimal digits to its numeric representation as follows:

```
> foldl (\x c -> 10*x+ord c-ord "0") 0 "123456";
123456
```

However, there are much easier and more general ways to convert between strings and Pure expressions. Specifically, `val` and `str` can be used to convert between any Pure value and its string representation:

```
> val "2*(3+4)"; str ans;
2*(3+4)
"2*(3+4)"
```

If you also want to evaluate the string representation of a Pure expression then `eval` is your friend:

```
> eval "2*(3+4)";
14
```

Two other convenient functions are `split` which breaks apart a string at a given delimiter

string, and `join` which concatenates a list of strings, interpolating the delimiter string between successive list elements:

```
> split " " "The quick brown fox";
["The","quick","brown","fox"]
> join ":" ans;
"The:quick:brown:fox"
```

If you don't need the intervening delimiters then you can also concatenate string lists simply with `strcat`:

```
> strcat ["The","quick","brown","fox"];
"Thequickbrownfox"
```

These operations are all implemented in an efficient way so that they run in linear time. (Note that the `string` conversion function we mentioned above is in fact just `strcat` on lists of strings, but it also works with other aggregates such as vectors of strings.)

For more elaborate needs there's also a suite of functions for doing regular expression matching in the `regex` module, and the `system` module provides the usual facilities for reading and writing strings from/to text files and the terminal, as well as the `printf` and `scanf` family of functions which are used to print and parse strings according to a given format string. These are all explained in detail in the *Pure Library Manual*.

1.5.12 List Comprehensions

List comprehensions are Pure's main workhorse for generating and processing all kinds of list values. You can think of them as a combination of `map` and `filter` using a prettier syntax reminiscent of the way in which sets are commonly specified in mathematics. List comprehensions are in fact just syntactic sugar, so anything that can be done with them can also be accomplished with Pure's generic list functions; but often they are much easier to write and understand.

In the simplest case, list comprehensions are just a shorthand for `map` with lambdas:

```
> [2*x-1 | x = 1..10];
[1,3,5,7,9,11,13,15,17,19]
```

This can be read aloud as "the list of all `2*x-1` for which `x` runs through the list `1..10`". The part `x = 1..10` is called a **generator clause**. The comprehension binds `x` to each member of the list `1..10` in turn and evaluates the target expression `2*x+1` in the context of this binding. This is equivalent to the following `map` expression:

```
> map (\x->2*x-1) (1..10);
[1,3,5,7,9,11,13,15,17,19]
```

List comprehensions may also involve **filter clauses**: predicates which determine the elements that are to be included in the result list.

```
> [2*x-1 | x = 1..10; x mod 3];
[1,3,7,9,13,15,19]
```

This can be read as "the list of all `2*x-1` for which `x` runs through `1..10` and for which `x mod 3` is non-zero" (which means that `x` is not a multiple of `3`). It is roughly equivalent to:

```
> map (\x->2*x-1) (filter (\x->x mod 3) (1..10));
[1,3,7,9,13,15,19]
```

List comprehensions can also draw values from other kinds of aggregates such as strings and matrices, but the result is always a list:

```
> [x-1 | x = "IBM"];
["H","A","L"]
> [1/x | x = {1,2,3;4,5,6}; ~x mod 2];
[0.5,0.25,0.166666666666667]
```

List comprehensions can have as many generator and filter clauses as you want. The clauses are considered in left-to-right order so that later clauses may refer to any variables introduced in earlier generator clauses. E.g., here's how you can generate the list of all pairs `(i,j)` with `1<=i<=j<=5` such that `i+j` is even:

```
> [i,j | i = 1..5; j = i..5; ~(i+j) mod 2];
[(1,1),(1,3),(1,5),(2,2),(2,4),(3,3),(3,5),(4,4),(5,5)]
```

The left-hand side of a generator clause can be an arbitary pattern, which is useful if you need to peek at the list elements to see what's inside. For instance, let's take the previous result and check that the sums of the number pairs are in fact all even:

```
> [i+j | i,j = ans];
[2,4,6,4,6,6,8,8,10]
```

Generator clauses involving patterns also act as filters; unmatched elements are filtered out automatically:

```
> [i+j | i,j = ["to be ignored",(1,1),(2,2),3]];
[2,4]
```

List comprehensions can also be nested to an arbitrary depth. For instance, we may rewrite the "even sums" comprehension from above as follows, in order to group the pairs into sublists for each value of `i`:

```
> [[i,j | j = i..5; ~(i+j) mod 2] | i = 1..5];
[[(1,1),(1,3),(1,5)],[(2,2),(2,4)],[(3,3),(3,5)],[(4,4)],[(5,5)]]
```

A notorious example is the following recursive algorithm which implements a variation of Erathosthenes' classical prime sieve. (This method is actually rather slow and thus not suitable for computing large primes, but we're not concerned with that here.)

```
primes n        = sieve (2..n) with
  sieve []      = [];
```

```
  sieve (p:qs)  = p : sieve [q | q = qs; q mod p];
end;
```

Note that the sieve recursively filters out the multiples of the current front element `p` of the list, which, by virtue of the construction, is always a prime number. The result is the list of all primes up to `n`:

```
> primes 100;
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

List comprehensions are also a useful device to organize backtracking searches. For instance, here's an algorithm for the n queens problem, which returns the list of all placements of `n` queens on an `n` x `n` board (encoded as lists of `n` pairs `(i,j)` with `i = 1..n`), so that no two queens hold each other in check:

```
queens n       = search n 1 [] with
  search n i p = [reverse p] if i>n;
               = cat [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = ~any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

1.5.13 Lazy Evaluation and Streams

As already mentioned, lists can also be evaluated in a "lazy" fashion, by just turning the tail of a list into a future. This special kind of list is also called a **stream**. Streams enable you to work with infinite lists (or finite lists which are so huge that you would never want to keep them in memory in their entirety). E.g., here's one way to define the infinite stream of all Fibonacci numbers:

```
> let fibs = fibs 0L 1L with fibs a b = a : fibs b (a+b) & end;
> fibs;
0L:#<thunk 0xb5d54320>
```

Note the `&` on the tail of the list in the definition of the local `fibs` function. This turns the result of `fibs` into a stream, which is required to prevent the function from recursing into samadhi. Also note that we work with bigints in this example because the Fibonacci numbers grow quite rapidly, so with machine integers the values would soon start wrapping around to negative integers.

Streams like these can be worked with in pretty much the same way as with lists. Of course, care must be taken not to invoke "eager" operations such as `#` (which computes the size of a list) on infinite streams, to prevent infinite recursion. However, many list operations work with infinite streams just fine, and return the appropriate stream results. E.g., the `take` function (which retrieves a given number of elements from the front of a list) works with streams just as well as with "eager" lists:

```
> take 10 fibs;
0L:#<thunk 0xb5d54350>
```

Hmm, not much progress there, but that's just how streams work (or rather they don't, they're lazy bums indeed!). Nevertheless, the stream computed with `take` is in fact finite and we can readily convert it to an ordinary list, forcing its evaluation:

```
> list (take 10 fibs);
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L]
```

An alternative way to achieve this is to cut a "slice" from the stream:

```
> fibs!!(0..10);
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L,55L]
```

Note that since we bound the stream to a variable, the already computed prefix of the stream has been memoized, so that this portion of the stream is now readily available in case we need to have another look at it later. By these means, possibly costly reevaluations are avoided, trading memory for execution speed:

```
> fibs;
0L:1L:1L:2L:3L:5L:8L:13L:21L:34L:55L:#<thunk 0xb5d54590>
```

The prelude also provides some convenience operations for generating stream values. Infinite arithmetic sequences are specified using `inf` or `-inf` to denote an upper (or lower) infinite bound for the sequence, e.g.:

```
> let u = 1..inf; let v = -1.0:-1.2..-inf;
> u!!(0..10); v!!(0..10);
[1,2,3,4,5,6,7,8,9,10,11]
[-1.0,-1.2,-1.4,-1.6,-1.8,-2.0,-2.2,-2.4,-2.6,-2.8,-3.0]
```

Other useful stream generator functions are `iterate`, which keeps applying the same function over and over again, `repeat`, which just repeats its argument forever, and `cycle`, which cycles through the elements of the given list:

```
> iterate (*2) 1!!(0..10);
[1,2,4,8,16,32,64,128,256,512,1024]
> repeat 1!!(0..10);
[1,1,1,1,1,1,1,1,1,1,1]
> cycle [0,1]!!(0..10);
[0,1,0,1,0,1,0,1,0,1,0]
```

Moreover, list comprehensions can draw values from streams and return the appropriate stream result:

```
> let rats = [m,n-m | n=2..inf; m=1..n-1; gcd m (n-m) == 1]; rats;
(1,1):#<thunk 0xb5d54950>
> rats!!(0..10);
[(1,1),(1,2),(2,1),(1,3),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5),(5,1)]
```

We can also rewrite our prime sieve so that it generates the infinite stream of *all* prime numbers:

```
all_primes      = sieve (2..inf) with
  sieve (p:qs)  = p : sieve [q | q = qs; q mod p] &;
end;
```

Note that we can omit the empty list case of `sieve` here, since the sieve now never becomes empty. Example:

```
> let P = all_primes;
> P!!(0..20);
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73]
> P!299;
1987
```

You can also just print the entire stream. This will run forever, so hit `Ctrl-c` when you get bored:

```
> using system;
> do (printf "%d\n") all_primes;
2
3
5
  ...
```

It's also possible to convert an ordinary list to a stream:

```
> stream (1..10);
1:#<thunk 0x7f2692a0f138>
```

This may seem like a silly thing to do, because the original list is already fully known beforehand. But this transformation allows us to traverse the list in a lazy fashion, which can be useful if the list is employed in a list comprehension or processed by functions such as `cat` and `map`. For instance, we can use this to rewrite the `fringe` function from The Same-Fringe Problem so that it calculates the fringe in a lazy fashion:

```
lazyfringe t = if listp t then catmap lazyfringe (stream t) else [t];
```

Recall that the fringe of a tree is the list of its leaves in left-to-right order. The tree itself is represented as a nested list, to which `lazyfringe` applies `stream` recursively so that the fringe becomes a stream whose elements are only produced on demand:

```
> lazyfringe [[a,b],c,[[d]],e,[f,[[g,h]]]];
a:#<thunk 0x7f127fc1f090>
> list ans;
[a,b,c,d,e,f,g,h]
```

Hence a simple syntactic equality check now suffices to solve the same-fringe problem in an efficient way. For instance, consider the following sample trees from The Same-Fringe Problem:

```
let t1 = [[a,b],c,[[d]],e,[f,[[g,h]]]];
let t2 = [a,b,c,[[d],[],e],[f,[g,[h]]]];
let t3 = [[a,b],d,[[c]],e,[f,[[g,h]]]];
```

Let's also bind the fringes to some variables so that we can check which parts actually get evaluated:

```
let l1 = lazyfringe t1;
let l2 = lazyfringe t2;
let l3 = lazyfringe t3;
```

Now comparing `l3` and `l2` we get:

```
> l3 === l2; l3; l2;
0
a:b:d:#<thunk 0x7fd308116178>
a:b:c:#<thunk 0x7fd308116060>
```

As you can see, the two fringes were only constructed as far as needed to decide that they differ. Of course, if we compare `l1` and `l2` then the fringes will still be fully constructed before we find that they're equal:

```
> l1 === l2; l1; l2;
1
[a,b,c,d,e,f,g,h]
[a,b,c,d,e,f,g,h]
```

But this doesn't really matter if we construct the fringes as temporary values, as in:

```
> fringe t1 === fringe t2;
```

Now only the parts of the fringes are in memory which are currently under scrutiny as the '`===`' operator passes over them; the prefixes which have already been found to be equal can be garbage-collected immediately. Moreover, the '`===`' operator is tail-recursive so that the entire equality test can be executed in constant stack space. This gives us an easier way to solve the same-fringe problem which has pretty much the same benefits as our earlier solution using continuations. The latter might still be considered more elegant, because it works without actually constructing the fringes at all. But the solution using lazy evaluation is certainly much simpler.

1.5.14 Matrices and Vectors

Pure has a versatile matrix data structure offering compact storage and efficient random access to its members. Pure matrices work pretty much like in MATLAB or Octave, except that indexes are zero-based and elements are stored in C's row-major rather than Fortran's column-major format. They are also binary-compatible with the GNU Scientific Library (GSL) so that they can readily be passed to GSL functions for doing numeric calculations.

Pure offers a number of basic matrix operations, such as matrix construction, pattern matching, indexing, slicing, as well as getting the size and dimensions of a matrix. It does *not* supply built-in support for matrix arithmetic and other linear algebra algorithms, but it's easy to roll your own if desired, as we'll see below. (Usually this won't offer the same performance as the GSL and other carefully optimized C and Fortran routines, however. So if you need to do some heavy-duty number crunching then you might want to take a look at

the pure-gsl module available at the Pure website, which is an ongoing project to make the GSL functions available in Pure.)

Matrices are denoted using curly braces in Pure:

```
> let x = {1,2,3;4,5,6}; x;
{1,2,3;4,5,6}
```

Note that the semicolon is used to separate different rows, while the elements inside each row are separated with commas. Thus the above denotes a 2x3 matrix (2 rows, 3 columns). The `dim` function lets you check the dimensions, while the '`#`' operator gives the total number of elements:

```
> dim x; #x;
2,3
6
```

There's no separate data type for vectors; row and column vectors are simply represented as 1 x n and n x 1 matrices, respectively:

```
> dim {1,2,3}; dim {1;2;3};
1,3
3,1
```

Singleton and empty matrices can be denoted as follows:

```
> dim {1}; dim {};
1,1
0,0
```

The `transpose` function turns columns into rows and vice versa; in particular, you can also use this to convert between row and column vectors:

```
> transpose x;
{1,4;2,5;3,6}
> transpose {1,2,3}; transpose {1;2;3};
{1;2;3}
{1,2,3}
```

Note that matrices are immutable in Pure, so matrix functions like `transpose` always return a *new* matrix, leaving the original matrix unchanged. (If you need to modify matrices in-place for efficiency, then you can use the GSL or other C or Fortran functions.)

You can change the dimensions of a matrix with the `redim` function, provided that the size stays the same. So, for instance, we can turn the matrix `x` into a row vector as follows:

```
> redim (1,6) x;
{1,2,3,4,5,6}
```

Again, this doesn't change the original matrix, but returns a new matrix with the same contents and the requested dimensions. This operation also allows you to change the dimensions of an empty matrix which, as we've seen above, has dimensions `0,0` by default. Of course, this requires that either the number of rows or columns is still zero. For instance:

```
> redim (3,0) {};
{}
> dim ans;
3,0
```

Another way to do this is to just construct a zero matrix with zero rows or columns directly, see below. (Note that these different kinds of empty matrices are needed to represent the corner cases. E.g., a linear mapping from 3-dimensional vectors to the zero vector space corresponds to a 0x3 matrix which yields a 3x0 matrix when transposed.)

A number of other specific conversion operations are available, such as `rowvector` and `colvector` which convert a matrix to a row or column vector, respectively, or `diag` which extracts the main diagonal of a matrix:

```
> rowvector x;
{1,2,3,4,5,6}
> colvector x;
{1;2;3;4;5;6}
> diag x;
{1,5}
```

You can also extract the rows and columns of a matrix, which yields a list of the corresponding row and column vectors, respectively:

```
> rows x; cols x;
[{1,2,3},{4,5,6}]
[{1;4},{2;5},{3;6}]
```

There are a number of other operations which convert between matrices and different kinds of aggregates; please check the *Matrix Functions* section in the *Pure Library Manual* for details.

Element access uses the index operator '`!`'. You can either specify a pair `(i,j)` of row and column indices, or a single index `i` which treats the entire matrix as a single vector in row-major order:

```
> x!(0,2);
3
> x!3;
4
```

Slicing is done with the '`!!`' operator. The index range can be specified in different ways. First, a pair of lists of row and column indices cuts a rectangular slice from the matrix:

```
> x!!(0..1,1..2);
{2,3;5,6}
```

Second, a pair of a list and a row or column index cuts slices from individual rows or columns:

```
> x!!(0,1..2); x!!(0..1,2);
{2,3}
{3;6}
```

Third, a list of pairs of row and column indices, or a list of element indices gives a row vector with all the corresponding elements:

```
> x!![(0,2),(1,2)];
{3,6}
> x!!(2..3);
{3,4}
```

While most of the slices above are contiguous (a case which the prelude optimizes for), you can also specify indices in any order, possibly with duplicates. So we may not only cut submatrix slices, but also permute and/or copy rows and columns of a matrix along the way:

```
> x!!([1,0,1],0..2);
{4,5,6;1,2,3;4,5,6}
```

Matrices can also be constructed from submatrices by arranging the submatrices in rows or columns. In fact, the curly braces accept any combination of submatrices and scalars, provided that all dimensions match up:

```
> {1,{2,3};{4,5},6};
{1,2,3;4,5,6}
> {{1;4},{2,3;5,6}};
{1,2,3;4,5,6}
> {{1;2;3},{4;5;6}};
{1,4;2,5;3,6}
```

The end result *must* be a rectangular matrix, however, otherwise you'll get an exception indicating a submatrix whose dimensions don't match:

```
> {1,{2,3};{4,5}};
<stdin>, line 24: unhandled exception 'bad_matrix_value {4,5}'
while evaluating '{1,{2,3};{4,5}}'
```

This "splicing" of submatrices is especially useful when doing linear algebra, where matrices are often composed from smaller "block matrices" or vectors; we'll see an example of this later. (Sometimes this behaviour also gets in the way, and thus there are ways to disable it; see Symbolic Matrices below.)

Pure actually provides several different types of **numeric matrices**, which correspond to the different GSL matrix types for integer, floating point and complex numbers. (Note that complex numbers aren't a built-in data type in Pure, but there are ways to specify this kind of numbers and perform calculations with them; see the `math` module for details.) Which type of matrix is created by the curly braces depends on the element types. Homogeneous matrices which contain only `int`, `double` or `complex` values yield the corresponding type of GSL matrix. Matrices can also hold any other type of Pure value or an arbitrary mix of values, in which case they become **symbolic matrices**; we'll discuss these later.

The functions `imatrix`, `dmatrix` and `cmatrix` can be used to convert between the different kinds of numeric matrices. For instance:

```
> dmatrix {1,2,3;4,5,6};
{1.0,2.0,3.0;4.0,5.0,6.0}
> imatrix ans;
{1,2,3;4,5,6}
> cmatrix ans;
{1.0+:0.0,2.0+:0.0,3.0+:0.0;4.0+:0.0,5.0+:0.0,6.0+:0.0}
> dmatrix ans;
{1.0,0.0,2.0,0.0,3.0,0.0;4.0,0.0,5.0,0.0,6.0,0.0}
```

(Note that the latter conversion turns a complex into a double matrix, interleaving the real and imaginary parts of the original matrix.)

The same functions can also be used to construct zero matrices with given dimensions:

```
> imatrix (2,3);
{0,0,0;0,0,0}
> dmatrix (2,2);
{0.0,0.0;0.0,0.0}
> cmatrix (1,1);
{0.0+:0.0}
```

As already mentioned, this also gives you a direct way to create empty matrices with different dimensions. For instance:

```
> imatrix (0,3); dim ans;
{}
0,3
```

The prelude offers matrix versions of the common list operations like `map`, `foldl`, `zip` etc., which provide a way to implement common matrix operations. E.g., multiplying a matrix `x` with a scalar `a` amounts to mapping the function `(a*)` to `x`, which can be done as follows:

```
> type scalar x = ~matrixp x;
> a::scalar * x::matrix = map (a*) x;
> 2*{1,2,3;4,5,6};
{2,4,6;8,10,12}
```

Note that the `matrix` type tag or the `matrixp` predicate can be used to restrict a variable to matrix values. (The prelude provides a few other types and corresponding predicates for various specific kinds of matrices, see the *Pure Library Manual* for details.) In addition, we also introduced a convenience type `scalar` for non-matrix values here, so that we can distinguish scalar from matrix multiplication which will be discussed below.

Matrix addition and other element-wise operations can be realized using `zipwith`, which combines corresponding elements of two matrices using a given binary function:

```
> x::matrix + y::matrix = zipwith (+) x y if dim x == dim y;
> {1,2,3;4,5,6}+{1,2,1;3,2,3};
{2,4,4;7,7,9}
```

Another way to define matrix functions in Pure is to employ a **matrix pattern**. The Pure language has built-in support for these, so that they work like the other kinds of patterns

we've already encountered. For instance, to compute the dot product of two 2D vectors, you may write something like:

```
> {x1,y1}*{x2,y2} = x1*x2+y1*y2;
> {2,3}*{1,4};
14
```

Or, to compute the determinant of a 2x2 matrix:

```
> det {a,b;c,d} = a*d-b*c;
> det {1,2;3,4};
-2
```

These patterns are convenient if the dimensions of the involved matrices are small and known beforehand. If this isn't the case then it's better to use **matrix comprehensions** instead, which work with arbitrary dimensions and make it easy to express many simple kinds of algorithms which would typically be done using `for` loops in conventional programming languages.

Matrix comprehensions work pretty much like list comprehensions, but with a special twist: if values are drawn from lists then the generator clauses alternate between row and column generation. (More precisely, the last generator, which varies most quickly, yields a row, the next-to-last one a column of these row vectors, and so on.) This makes matrix comprehensions resemble customary mathematical notation very closely. For instance, here is how we can define an operation to create a square identity matrix of a given dimension (note that the `i==j` term is just a Pure idiom for the Kronecker symbol):

```
> eye n = {i==j | i = 1..n; j = 1..n};
> eye 3;
{1,0,0;0,1,0;0,0,1}
```

Of course, matrix comprehensions can also draw values from other matrices instead of lists. In this case the block layout of the component matrices is preserved. For instance:

```
> {x,y | x = {1,2}; y = {a,b;c,d}};
{(1,a),(1,b),(2,a),(2,b);(1,c),(1,d),(2,c),(2,d)}
```

Note that a matrix comprehension involving filters may fail because the filtered result isn't a rectangular matrix any more. E.g., `{2*x|x={1,2,3,-4};x>0}` works, as does `{2*x|x={-1,2;3,-4};x>0}`, but `{2*x|x={1,2;3,-4};x>0}` doesn't because the rows of the result matrix have different lengths.

As a slightly more comprehensive example (no pun intended!), here is a definition of matrix multiplication in Pure:

```
x::matrix * y::matrix = {dot u v | u = rows x; v = cols y} with
  dot u v = foldl (+) 0 $ zipwith (*) u (rowvector v);
end if m==n when _,m = dim x; n,_ = dim y end;
```

The basic building block in this example is the dot product of two vectors, which is defined as a local function. The matrix product is obtained by simply calculating the dot product of all the rows of `x` with all the columns of `y`. To make this work, the rows of `x` should be the

same length as the columns of y, we check this condition in the guard of the rule. Let's give it a try:

```
> {1,0;0,1}*{1,2;3,4};
{1,2;3,4}
> {0,1;1,0}*{1,2;3,4};
{3,4;1,2}
> {0,1;1,0;1,1}*{1,2,3;4,5,6};
{4,5,6;1,2,3;5,7,9}
> {1,2;3,4}*{1;1};
{3;7}
```

Well, that was easy. So let's take a look at a more challenging example, Gaussian elimination, which can be used to solve systems of linear equations. The algorithm brings a matrix into "row echelon" form, a generalization of triangular matrices. The resulting system can then be solved quite easily using back substitution.

Here is a Pure implementation of the algorithm. Note that the real meat is in the pivoting and elimination step (`step` function) which is iterated over all columns of the input matrix. In each step, `x` is the current matrix, `i` the current row index, `j` the current column index, and `p` keeps track of the current permutation of the row indices performed during pivoting. The algorithm returns the updated matrix `x`, row index `i` and row permutation `p`.

```
gauss_elimination x::matrix = p,x
when n,m = dim x; p,_,x = foldl step (0..n-1,0,x) (0..m-1) end;

// One pivoting and elimination step in column j of the matrix:
step (p,i,x) j
= if max_x==0 then p,i,x
  else
    // updated row permutation and index:
    transp i max_i p, i+1,
    {// the top rows of the matrix remain unchanged:
     x!!(0..i-1,0..m-1);
     // the pivot row, divided by the pivot element:
     {x!(i,l)/x!(i,j)                 | l=0..m-1};
     // subtract suitable multiples of the pivot row:
     {x!(k,l)-x!(k,j)*x!(i,l)/x!(i,j) | k=i+1..n-1; l=0..m-1}}
when
  n,m = dim x; max_i, max_x = pivot i (col x j);
  x = if max_x>0 then swap x i max_i else x;
end with
  pivot i x       = foldl max (0,0) [j,abs (x!j)|j=i..#x-1];
  max (i,x) (j,y) = if x<y then j,y else i,x;
end;
```

Please refer to any good textbook on numerical mathematics for a closer description of the algorithm. But here is a brief rundown of what happens in each elimination step: First we find the pivot element in column `j` of the matrix. (We're doing partial pivoting here, i.e., we only look for the element with the largest absolute value in column `j`, starting at row `i`. That's usually good enough to achieve numerical stability.) If the pivot is zero then we're done (the rest of the pivot column is already zeroed out). Otherwise, we bring it into the

pivot position (swapping row `i` and the pivot row), divide the pivot row by the pivot, and subtract suitable multiples of the pivot row to eliminate the elements of the pivot column in all subsequent rows. Finally we update `i` and `p` accordingly and return the result.

In order to complete the implementation, we still need the following little helper functions to swap two rows of a matrix (this is used in the pivoting step) and to apply a transposition to a permutation (represented as a list):

```
swap x i j = x!!(transp i j (0..n-1),0..m-1) when n,m = dim x end;
transp i j p = [p!tr k | k=0..#p-1]
with tr k = if k==i then j else if k==j then i else k end;
```

Finally, let us define a convenient print representation of double matrices a la Octave (the meaning of the `__show__` function is explained in Pretty-Printing):

```
using system;
__show__ x::matrix
= strcat [printd j (x!(i,j))|i=0..n-1; j=0..m-1] + "\n"
with printd 0 = sprintf "\n%10.5f"; printd _ = sprintf "%10.5f" end
when n,m = dim x end if dmatrixp x;
```

Example:

```
> let x = dmatrix {2,1,-1,8; -3,-1,2,-11; -2,1,2,-3};
> x; gauss_elimination x;
   2.00000   1.00000  -1.00000   8.00000
  -3.00000  -1.00000   2.00000 -11.00000
  -2.00000   1.00000   2.00000  -3.00000
[1,2,0],
   1.00000   0.33333  -0.66667   3.66667
   0.00000   1.00000   0.40000   2.60000
   0.00000   0.00000   1.00000  -1.00000
```

1.5.15 Symbolic Matrices

As already mentioned, matrices may contain not just numbers but any kind of Pure values, in which case they become *symbolic* matrices. For instance:

```
> {1,2.0,3L;a,b,c};
{1,2.0,3L;a,b,c}
```

The `smatrixp` predicate gives you a quick way to check whether a matrix is a symbolic one:

```
> smatrixp ans;
1
```

Note that this may not always be obvious. For instance, you can use the `smatrix` function to explicitly convert a numeric matrix:

```
> smatrix {1,2;3,4};
{1,2;3,4}
```

This still looks the same as the original matrix, but `smatrixp` reveals that it's in fact a symbolic matrix:

```
> smatrixp ans;
1
```

Also note that the empty matrix is by default a symbolic matrix, as are matrices containing bigints:

```
> smatrixp {};
1
> smatrixp {1L,2L;3L,4L};
1
```

However, you can easily convert these to a numeric type if needed, e.g.:

```
> dmatrix {1L,2L;3L,4L};
{1.0,2.0;3.0,4.0}
```

Symbolic matrices are a convenient data structure for storing arbitrary collections of values which provides fast random access to its members. In particular, they can also be nested, and thus multidimensional tensors or arrays of arbitrary dimension can be realized as nested symbolic vectors. However, you have to be careful when constructing such values, because the `{...}` construct normally combines submatrices to larger matrices. For instance:

```
> {{1,2},{3,4}};
{1,2,3,4}
```

One way to inhibit this splicing of the submatrices in a larger matrix is to use the quote operator (cf. The Quote):

```
> '{{1,2},{3,4}};
{{1,2},{3,4}}
```

Note that this result is really different from `{1,2;3,4}`. The latter is a 2x2 integer matrix, while the former is a symbolic vector a.k.a. 1x2 matrix whose elements happen to be two integer vectors. So a double index will be required to access the subvector elements:

```
> ans!0!1;
2
```

You can also match these values with a nested matrix pattern, e.g.:

```
> let {{a,b},{c,d}} = '{{1,2},{3,4}};
> a,b,c,d;
1,2,3,4
```

Unfortunately, the quote operator in fact inhibits evaluation of *all* embedded subterms which may be undesirable if the matrix expression contains arithmetic (as in `'{{1+1,2*3}}`), so this method works best for constant matrices. A more general way to create a symbolic vector of matrices is provided by the `vector` function from the prelude, which is applied to a list of the vector elements as follows:

```
> vector [{1,2},{3,4}];
{{1,2},{3,4}}
```

Calls to the `vector` function can be nested to an arbitrary depth to obtain higher-dimensional "arrays":

```
> vector [vector [{1,2}],vector [{3,4}]];
{{{1,2}},{{3,4}}}
```

This obviously becomes a bit unwieldy for higher dimensions, but Pure 0.56 and later provide the following shorthand notation:

```
> {|{1,2},{3,4}|};
{{1,2},{3,4}}
> {|{|{1,2}|},{|{3,4}|}|};
{{{1,2}},{{3,4}}}
```

This makes it much more convenient to denote nested vector values. Note that the `{| |}` construct doesn't use any special magic, it's just a standard outfix operator implemented as a Pure macro. For more details please check the description of the *non-splicing vector brackets* in the *Pure Library Manual*.

1.5.16 Record Data

Symbolic matrices also provide a means to represent simple record-like data, by encoding records as symbolic vectors consisting of "hash pairs" of the form `key => value`. This kind of data structure is very convenient to represent aggregates with lots of different components. Since the components of records can be accessed by indexing with key values, you don't have to remember which components are stored in which order, just knowing the keys of the required members is enough. In contrast, tuples, lists and other kinds of constructor terms quickly become unwieldy for such purposes.

The keys used for indexing the record data must be either symbols or strings, while the corresponding values may be arbitrary Pure values. The prelude provides some operations on these special kinds of matrices, which let you retrieve vector elements by indexing and perform non-destructive updates, see the *Record Functions* section in the *Pure Library Manual* for details. Here are a few examples which illustrate how to create records and work with them:

```
> let r = {x=>5, y=>12};
> recordp r, member r x;
1,1
> r!y; r!![y,x];
12
{12,5}
> insert r (x=>99);
{x=>99,y=>12}
> insert ans (z=>77);
{x=>99,y=>12,z=>77}
```

```
> delete ans z;
{x=>99,y=>12}
```

Records can also be nested:

```
> let r = {a => {b=>1,c=>2}, b => 2};
> r!a, r!b, r!a!b;
{b=>1,c=>2},2,1
```

Note the use of the "hash rocket" `=>` which denotes the `key=>value` associations in a record. The hash rocket is a constructor declared as an infix operator in the prelude, see the *Hash Pairs* section in the *Pure Library Manual* for details. There's one caveat here, however. Since neither '=>' nor '!' treat their key operand in a special way, you'll have to take care that the key symbols do not evaluate to something else, as might be the case if they are bound to a global or local variable or parameterless function:

```
> let u = 99;
> {u=>u};
{99=>99}
```

In the case of global variables and function symbols, you might protect the symbol with a quote (see The Quote):

```
> {'u=>u};
{u=>99}
> ans!'u;
99
```

However, even the quote doesn't save you from local variable substitution:

```
> {'u=>u} when u = 99 end;
{99=>99}
```

In such cases you'll either have to rename the local variable, or use the prelude function `val` to quote the symbol:

```
> {'u=>v} when v = 99 end;
{u=>99}
> {val "u"=>u} when u = 99 end;
{u=>99}
```

It's also possible to directly use strings as keys instead, which may actually be more convenient in some cases:

```
> let r = {"x"=>5, "y"=>12};
> keys r; vals r;
{"x","y"}
{5,12}
> update r "y" (r!"y"+1);
{"x"=>5,"y"=>13}
```

You can also mix strings and symbols as keys in the same record (but note that strings and symbols are always distinct, so `y` and `"y"` are really two different keys here):

```
> insert r (y=>99);
{"x"=>5,"y"=>12,y=>99}
```

As records are in fact just special kinds of matrices, the standard matrix operations can be used on record values as well. For instance, the matrix constructor provides an alternative way to quickly augment a record with a collection of new `key=>value` associations:

```
> let r = {x=>5, y=>12};
> let r = {r, x=>7, z=>3}; r;
{x=>5,y=>12,x=>7,z=>3}
> r!x, r!z;
7,3
> delete r x;
{x=>5,y=>12,z=>3}
> ans!x;
5
```

As the example shows, this may produce duplicate keys, but these are handled gracefully; indexing and updates will always work with the *last* association for a given key in the record. If necessary, you can remove duplicate entries from a record as follows; this will only keep the last association for each key:

```
> record r;
{x=>7,y=>12,z=>3}
```

In fact, the `record` operation not only removes duplicates, but also orders the record entries by keys. This produces a kind of normalized representation which is useful if you want to compare or combine two record values irrespective of the ordering of the fields. For instance:

```
> record {x=>5, y=>12} === record {y=>12, x=>5};
1
```

The `record` function can also be used to construct a normalized record directly from a list or tuple of hash pairs:

```
> record [x=>5, x=>7, y=>12];
{x=>7,y=>12}
```

Other matrix operations such as `map`, `foldl`, etc., and matrix comprehensions can be applied to records just as easily. This enables you to perform bulk updates of record data in a straightforward way. For instance, here's how you can define a function `maprec` which applies a function to all values stored in a record:

```
> maprec f = map (\(u=>v) -> u=>f v);
> maprec (*2) {x=>5,y=>12};
{x=>10,y=>24}
```

Another example: The following `ziprec` function collects pairs of values stored under common keys in two records (we also normalize the result here so that duplicate keys are always removed):

```
> ziprec x y = record {u=>(x!u,y!u) | u = keys x; member y u};
> ziprec {a=>3,x=>5,y=>12} {x=>10,y=>24,z=>7};
{x=>(5,10),y=>(12,24)}
```

Thus the full power of generic matrix operations is available for records, which turns them into a much more versatile data structure than records in conventional programming languages, which are usually limited to constructing records and accessing or modifying their components.

Note that since the values stored in records can be arbitrary Pure values, you can also have records with mutable components by making use of Pure's *expression references*. For instance:

```
> let r = {x=>ref 1,y=>ref 2}; maprec get r;
{x=>1,y=>2}
> put (r!x) 99; maprec get r;
99
{x=>99,y=>2}
```

Another interesting application of records are the "virtual method tables" used in object-oriented programming. Pure has a built-in `__locals__` macro which captures the environment of local functions at the point of the call and returns it as a list of hash pairs of function symbols and the corresponding closures. We can readily convert this into a record data structure which can be used as a virtual method table. For instance:

```
> record __locals__ with f x = x+1 end;
{f=>f}
> (ans!f) 99;
100
```

Here is a little helper macro that we can use to turn the virtual method table into an anonymous function which, when applied to a symbol, returns the appropriate closure:

```
def obj = (\x -> vt!x) when
  vt = record __locals__;
end;
```

Continuing our example from Local Functions and Variables, we can now implement the `point` object as follows:

```
point (x,y) = obj with
  coords () = get x,get y;
  move (dx,dy) = put x (get x+dx), put y (get y+dy);
end when
  x,y = ref x,ref y;
end;
```

Note that `obj` really needs to be implemented as a macro so that its body is inserted into the `point` function and the `_locals__` call is executed in the context of the local function environment there. (A macro is like a function which gets evaluated at compile time; see the Macros section for details.) Also note that we changed the `coords` "method" so that it takes a dummy parameter `()` now; this prevents premature evaluation of the closure. If `coords` was

a parameterless function then its value would be fixed at the time we construct the virtual method table, which is not what we want here.

Now we can write:

```
> let p = point (1,2);
> p coords ();
1,2
> p move (2,3);
3,5
> p coords ();
3,5
```

This provides us with an interesting way to represent stateful objects which works very much like object-oriented programming. What's still missing here is the inheritance of methods from other objects, but this can now be done by just combining virtual method tables using the record operations we've already discussed above; we leave this as an exercise for the interested reader.

1.5.17 The Quote

We've already seen some uses of the quote in previous examples, so let's have a closer look at it now. As described in Special Forms, the `quote` operation quotes an expression, so that it can be passed around and manipulated freely until its value is needed, in which case you can pass it to the `eval` function to obtain its value. For instance:

```
> let x = '(2*42+2^12); x;
2*42+2^12
> eval x;
4180.0
```

Lisp programmers will be well familiar with this operation which enables some powerful metaprogramming techniques. However, there are some notable differences to Lisp's quote. In particular, `quote` only inhibits the evaluation of global variables, *local* variables are substituted as usual:

```
> (\x -> '(2*x+1)) 99;
2*99+1
> foo x = '(2*x+1);
> foo 99; foo $ '(7/y);
2*99+1
2*(7/y)+1
> '(x+1) when x = '(2*3) end;
2*3+1
> '(2*42+2^n) when n = 12 end;
2*42+2^12
```

Local parameterless functions are treated in the same fashion:

```
> '(2*42+2^n) with n = 12 end;
2*42+2^12
```

Note that, in contrast, for global variables (and functions) we have:

```
> let n = 12;
> '(2*42+2^n);
2*42+2^n
```

This discrepancy may come as a surprise (or even annoyance) to real Lisp weenies, but it does have its advantages. As illustrated in the examples above, local variable substitution makes it easy to fill in the variable parts in a quoted "template" expression, without any need for an arguably complex tool like Lisp's "quasiquote". (But note that it is quite easy to define the quasiquote in Pure if you want it. See the Recursive Macros section for a simplified version; a full implementation can be found in the Pure library.)

If you do need to quote a symbol which is already being used as a local variable or function in the current context, you can do this by supplying the symbol as a string to the prelude function `val`:

```
> val "x"+x when x = 99 end;
x+99
```

Also note that while local functions are always substituted in a quoted expression, *applications* involving local functions can still be quoted:

```
> 'foo 99 with foo x = 2*x+1 end;
foo 99
> eval ans;
199
```

The quote also inhibits evaluation inside matrix expressions, including the "splicing" of embedded submatrices:

```
> '{1,2+3,2*3};
{1,2+3,2*3}
> '{1,{2,3},4};
{1,{2,3},4}
```

Special expressions (conditionals, lambda and the `case`, `when` and `with` constructs) can be quoted as well. But since these constructs cannot be directly represented at runtime, the quote actually produces some ordinary "placeholder" terms for these:

```
> '(x+1 when x = '(2*3) end);
x+1 __when__ [x-->'(2*3)]
> eval ans;
2*3+1
> '(2*42+(f 6 with f n = 2^(2*n) end));
2*42+(f 6 __with__ [f n-->2^(2*n)])
> eval ans;
4180.0
```

Note that these placeholders are in fact special built-in macros which reconstruct the special expression when evaluated. Moreover, special expressions are implicitly quoted when they occur on the left-hand side of an equation or as an argument of a "*`quoteargs`*" macro call.

This is often used to implement macros which manipulate these constructs as literals. For instance, the following macro swaps the arguments in a lambda:

```
> #! --quoteargs bar
> def bar (\x y -> z) = __eval__ ('(\y x -> z));
> show bar
def bar (__lambda__ [x,y] z) = __eval__ ('__lambda__ [y,x] z);
> baz = bar (\a b -> a-b);
> show baz
baz = \b a -> a-b;
> baz 2 3;
1
```

The Macros section explains in detail how this meta programming works.

1.6 Declarations

Pure is a very terse language by design. Usually you don't declare much stuff, you just define it and be done with it. However, there are a few constructs which let you declare symbols with special attributes and manage programs consisting of several source modules:

- symbol declarations determine "scope" and "fixity" of a symbol;
- `interface` declarations specify abstract data types;
- `extern` declarations specify external C functions;
- `using` clauses let you include other scripts in a Pure script;
- `namespace` declarations let you avoid name clashes and thereby make it easier to manage large programs consisting of many separate modules.

These are toplevel elements (cf. Toplevel):

```
item  ::=   symbol_decl | interface_decl | extern_decl
          | using_decl | namespace_decl
```

We defer the discussion of `extern` declarations to the C Interface section. The other kinds of declarations are described in the following subsections.

1.6.1 Symbol Declarations

Symbol declarations declare special attributes of a symbol, such as their scope (whether they are "public" or "private") and their fixity (for operator symbols). The syntax of these declarations is as follows:

```
symbol_decl  ::=  scope qualified_symbol+ ";"
                  | [scope] fixity qualified_symbol+ ";"
scope        ::=  "public" | "private"
fixity       ::=  "nonfix" | "outfix"
                  | ("infix"|"infixl"|"infixr"|"prefix"|"postfix") precedence
precedence   ::=  integer | "(" op ")"
```

Scope declarations take the following form:

```
public symbol ...;
private symbol ...;
```

This declares the listed symbols as public or private, respectively. Each symbol must either be an identifier or a sequence of punctuation characters. The latter kind of symbols *must* always be declared before use, whereas ordinary identifiers can be used without a prior declaration in which case they are declared implicitly and default to public scope, meaning that they are visible everywhere in a program. An explicit public declaration of ordinary identifiers is thus rarely needed (unless you want to declare symbols as members of a specific namespace, see Namespaces below). Symbols can also be declared private, meaning that the symbol is visible only in the namespace it belongs to. This is explained in more detail under Private Symbols in the Namespaces section below.

Note: The declared symbols may optionally be qualified with a namespace prefix, but since new symbols can only be created in the current namespace, the namespace prefix must match the current namespace (see Namespaces). Thus the namespace prefix isn't really needed, unless you want to declare a symbol which happens to be a reserved Pure keyword (cf. Lexical Matters). In this specific case, it will be necessary to use a qualified name so that the symbol isn't mistaken for a keyword.

Note that to declare several symbols in a single declaration, you can list them all with white-space in between. The same syntax applies to the other types of symbol declarations discussed below. (Commas are *not* allowed as delimiters here, as they may occur as legal symbol constituents in the list of symbols.) The `public` and `private` keywords can also be used as a prefix in any of the special symbol declarations discussed below, to specify the scope of the declared symbols (if the scope prefix is omitted, it defaults to `public`).

The following "fixity" declarations are available for introducing special operator symbols. This changes the way that these symbols are parsed and thus provides you with a limited means to extend the Pure language at the lexical and syntactical level.

```
infix level symbol ...;
infixl level symbol ...;
infixr level symbol ...;
prefix level symbol ...;
postfix level symbol ...;
```

Pure provides you with a theoretically unlimited number of different precedence levels for user-defined infix, prefix and postfix operators. Precedence levels are numbered starting at

0; larger numbers indicate higher precedence. (For practical reasons, the current implementation does require that precedence numbers can be encoded as 24 bit unsigned machine integers, giving you a range from 0 to 16777215, but this should be large enough to incur no real limitations on applications. Also, the operator declarations in the prelude have been set up to leave enough "space" between the "standard" levels so that you can easily sneak in new operator symbols at low, high or intermediate precedences.)

On each precedence level, you can declare (in order of increasing precedence) `infix` (binary non-associative), `infixl` (binary left-associative), `infixr` (binary right-associative), `prefix` (unary prefix) and `postfix` (unary postfix) operators. For instance, here is a typical excerpt from the prelude (the full table can be found in the *Prelude* section of the *Pure Library Manual*):

```
infix  1800 < > <= >= == ~= ;
infixl 2200 + - ;
infixl 2300 * / div mod ;
infixr 2500 ^ ;
prefix 2600 # ;
```

Note: Unary minus plays a special role in the syntax. Like in Haskell and following mathematical tradition, unary minus is the only prefix operator symbol which is also used as an infix operator, and is always on the same precedence level as binary minus, whose precedence may be chosen freely in the prelude. (The minus operator is the only symbol which gets that special treatment; all other operators must have distinct lexical representations.) Thus, with the standard prelude, `-x+y` will be parsed as `(-x)+y`, whereas `-x*y` is the same as `-(x*y)`. Also note that the notation `(-)` always denotes the binary minus operator; the unary minus operation can be denoted using the built-in `neg` function.

Instead of denoting the precedence by an explicit integer value, you can also specify an existing operator symbol enclosed in parentheses. Thus the following declaration gives the ++ operator the same precedence as +:

```
infixl (+) ++ ;
```

The given symbol may be of a different fixity than the declaration, but it must have a proper precedence level (i.e., it must be an infix, prefix or postfix symbol). E.g., the following declaration gives ^^ the same precedence level as the infix ^ symbol, but turns it into a postfix operator:

```
postfix (^) ^^ ;
```

Pure also provides unary outfix operators, which work like in Wm Leler's constraint programming language Bertrand. These can be declared as follows:

```
outfix left right ...;
```

Outfix operators let you define your own bracket structures. The operators must be given as pairs of matching left and right symbols (which must be distinct). For instance:

```
outfix |: :| BEGIN END;
```

After this declaration you can write bracketed expressions like `|:x:|` or `BEGIN foo, bar END`. These are always at the highest precedence level (i.e., syntactically they work like parenthesized expressions). Just like other operators, you can turn outfix symbols into ordinary functions by enclosing them in parentheses, but you have to specify the symbols in matching pairs, such as `(BEGIN END)`.

Pure also has a notation for "nullary" operators, that is, "operators without operands". These are used to denote special literals which simply stand for themselves. They are introduced using a `nonfix` declaration:

```
nonfix symbol ...;
```

For instance:

```
nonfix red green blue;
```

Semantically, nonfix symbols are a kind of "symbolic constants". However, it is important to note the difference to *defined* constants, which are symbols bound to a constant value by means of a `const` definition. In fact, there are some use cases where a symbol may be *both* a defined constant and a nonfix symbol, see Constant Definitions in the Caveats and Notes section for details.

Syntactically, nonfix symbols work just like ordinary identifiers, so they may stand whereever an identifier is allowed (no parentheses are required to "escape" them). However, just like other kinds of operators, they may also consist of punctuation (which isn't allowed in ordinary identifiers). The other difference to ordinary identifiers is that nonfix symbols are always interpreted as literals, even if they occur in a variable position on the left-hand side of a rule. So, with the above declaration, you can write something like:

```
> foo x = case x of red = green; green = blue; blue = red end;
> map foo [red,green,blue];
[green,blue,red]
```

Thus nonfix symbols are pretty much like nullary constructor symbols in languages like Haskell. Non-fixity is just a syntactic attribute, however. Pure doesn't enforce that such values are irreducible, so you can still write a "constructor equation" like the following:

```
> red = blue;
> map foo [red,green,blue];
[blue,blue,blue]
```

Examples for all types of symbol declarations can be found in the *prelude* which declares a bunch of standard (arithmetic, relational, logical) operator symbols as well as the list and pair constructors ':' and ',', and a few nonfix symbols (`true` and `false`, as well as different kinds of exceptions).

1.6.2 Interface Types

Besides the "concrete" types already described in the Type Rules section, Pure provides another, more abstract way to characterize a type through the collection of operations it supports. These **interface types** work pretty much like in Google's Go programming language. They provide a safe form of Duck typing in which the operations available on a type are stated explicitly, and hence members of the type are always known to provide all of the listed operations.

An interface declaration gives the type name along with a collection of patterns, the so-called **signature** which specifies the manifest operations of the type:

```
interface_decl  ::=   "interface" qualified_identifier
                      "with" interface_item* "end" ";"
interface_item  ::=   pattern ";"
                      | "interface" qualified_identifier ";"
```

Interfaces thus consist of two kinds of items:

- The patterns, which indicate which operations are supported by the type, and which arguments they expect. This may be anything that can occur as the left-hand side of an ordinary function definition, cf. General Rules.
- The name of another interface type. This causes the signature of the named interface type to be included in the interface type being defined, which effectively turns the new interface type into a subtype of the existing one.

The gist of an interface is in its patterns, more precisely: in the pattern variables which have the name of the interface as a type tag. The precise meaning of the patterns is as follows:

- The patterns are matched against the left-hand sides of ordinary function definitions. If a left-hand side matches, any argument pattern substituted for a variable tagged with the interface type becomes a "candidate pattern" of the type.
- The type consists of all candidate patterns which can be matched by some candidate pattern of *each* interface function. That is, candidate patterns which are only supported by some but not all of the interface functions, are eliminated.
- Finally, all trivial candidate patterns (`x` where `x` is just a variable without any type tag, which thus matches *any* value) are eliminated as well.

Interface patterns often take a simple form like the following,

```
interface foo with foo x::foo y z; end;
```

specifying the number of arguments of the interface function along with the position of the interface type argument. However, general patterns are permitted, in order to further restrict the left-hand sides of the function definitions to be taken into consideration. Specifically, note that type tags other than the interface type must always be matched *literally* on the left-hand sides of equations. Thus,

```
interface foo with foo x::foo y::int; end;
```

matches any rule of the form

```
foo x y::int = ...;
```

but *not*:

```
foo x 0 = ...;
foo x y::bar = ...;
```

(unless `bar` happens to be an alias of the `int` type, of course). In such cases it is necessary to explicitly add these patterns to the interface if you want them to be included.

Interface patterns may contain the interface type tag any number of times, yielding candidate patterns for each occurrence of the interface type tag in the pattern. For instance, here is a quick way to determine the type of all "addable" data structures in the prelude (this uses the interactive `show interface` command to list the patterns actually matched by an interface type, cf. The show Command):

```
> interface addable with x::addable + y::addable; end;
> show interface addable
type addable x::int;
type addable x::double;
type addable x::bigint;
type addable s::string;
type addable [];
type addable xs@(_:_);
```

On the other hand, interfaces may also contain "static" patterns which do not include the interface type as a tag at all, such as:

```
interface foo with bar x::bar y; end;
```

These do not contribute anything to the candidate patterns of the type, but do restrict the type just like the other patterns, in that the type will be empty unless the static patterns are all "implemented". In the example above, this means that the `foo` type will be empty unless the `bar` function is defined and takes an element of the `bar` type as its first argument.

An interface may also be empty, in which case it matches *any* value. Thus,

```
interface any with end;
```

is just a fancy way to define the type:

```
type any _;
```

Interfaces can be composed in a piecemeal fashion, by adding more interface patterns. Thus,

```
interface foo with foo x::foo; end;
interface foo with bar x::foo; end;
```

is equivalent to:

```
interface foo with foo x::foo; bar x::foo; end;
```

It is also possible to include one interface in another, which effectively establishes a subtype relationship. For instance, here's yet another way to define the `foo` interface above:

```
interface bar with
  bar x::bar;
end;

interface foo with
  foo x::foo;
  interface bar;
end;
```

This has the effect of including the signature of `bar` in `foo` (while renaming the interface type tags in the `bar` signature accordingly):

```
> show foo
interface foo with
  foo x::foo;
  bar x::foo;
end;
```

Note: Including interfaces is a static operation. Only the interface patterns known at the point of inclusion become part of the including interface; refining the included interface later has no effect on the set of included patterns. In particular, this also prevents circular interface definitions.

When composing interfaces in this fashion, it is easy to end up with duplicate interface patterns from various sources. The compiler removes such duplicates, even if they only match up to the renaming of variables. For instance:

```
> show bar foo
interface bar with
  bar x::bar;
end;
interface foo with
  foo x::foo;
  bar x::foo;
end;
> interface baz with
>   interface foo; interface bar;
>   foo y::baz;
> end;
> show baz
interface baz with
  foo x::baz;
  bar x::baz;
end;
```

Also note that, despite the obvious similarities between interfaces and classes in object-

oriented programming, they are really different things. The former are essentially just signatures of functions living elsewhere, whereas the latter also include data layouts and method implementations. More on the similarities and differences of interfaces and classes can be found in the Go FAQ.

Let's now take a look at the example of a stack data structure to see how this all works in practice:

```
interface stack with
  push s::stack x;
  pop s::stack;
  top s::stack;
end;
```

Note the use of the type tag `stack` in the operation patterns, which marks the positions of `stack` arguments of the interface operations. The interface tells us that a stack provides three operations `push`, `pop` and `top` which each take a stack as their first argument; also, `push` takes two arguments, while `pop` and `top` just take a single (stack) argument.

This information is all that the compiler needs to figure out which terms are members of the `stack` data type. To these ends, the compiler looks at existing definitions of `push`, `pop` and `top` and extracts the patterns for arguments marked with the `stack` tag in the interface. The `stack` patterns implemented by *all* of the interface operations make up the `stack` type; i.e., the members of the type are all the instances of these patterns.

Right now our `stack` type doesn't have any members, because we didn't implement the interface operations yet, so let's do this now. For instance, to implement stacks as lists, we might define:

```
push xs@[] x | push xs@(_:_) x = x:xs;
pop (x:xs) = xs;
top (x:xs) = x;
```

This is also known as "instantiating" the type. In addition, we will need an operation to create an initial stack value. The following will do for our purposes:

```
stack xs::list = xs;
```

This yields a stack with the given initial contents. Let's give it a go:

```
> top (push (stack []) 99);
99
```

Looks good so far. We can also check the actual definition of the type in terms of its `type` rules using the `show interface` command:

```
> show interface stack
type stack xs@(_:_);
```

Wait, something seems to be wrong there. The empty list pattern of the `push` function is missing, where did it go? Let's restart the interpreter with warnings enabled (`-w`) and retype the above definitions. The compiler then tells us:

```
> show interface stack
warning: interface 'stack' may be incomplete
warning: function 'pop' might lack a rule for 'xs@[]'
warning: function 'top' might lack a rule for 'xs@[]'
type stack xs@(_:_);
```

See? A pattern is only considered part of the type if it is supported by *all* the interface operations. Since the `pop` and `top` operations don't have any rules for empty list arguments, empty lists are excluded from the type. We can fix this quite easily by adding the following "error rules" which handle this case:

```
> pop [] = throw "empty stack";
> top [] = throw "empty stack";
> show interface stack
type stack xs@[];
type stack xs@(_:_);
```

This looks fine now, so let's see how we can put our new stack data structure to good use. Operations on the type are defined as usual, employing `stack` as a type tag for stack arguments so that we can be sure that the `push`, `pop` and `top` operations are all supported. For instance, let's implement a little RPN ("Reverse Polish Notation") calculator:

```
rpn xs::stack ops::list = foldl (call []) xs ops with
  call ys xs op = push xs (foldl ($) op ys) if nargs op<=#ys;
                = call (top xs:ys) (pop xs) op otherwise;
end;
```

This takes an initial stack `xs` and a list `ops` of operands and operations as inputs and returns the resulting stack after processing `ops`. Examples:

```
> rpn (stack []) [10,4,3,(+),2,(*),(-)];
[-4]
> using math;
> rpn (stack []) [1,2,ln,(/)];
[1.44269504088896]
> rpn (stack []) [4,1,atan,(*)];
[3.14159265358979]
> rpn (stack []) [2,(*)];
<stdin>, line 5: unhandled exception '"empty stack"' while evaluating
'rpn (stack []) [2,(*)]'
```

Ok, this is all very nice, but it seems that so far we haven't done much more than we could have achieved just as easily with plain lists instead. So what are the benefits of having an interface type?

First, an interface provides a fair amount of **safety**. As long as we stick to the interface functions, we can be sure that the data is capable of carrying out the requested operations. At the same time, the interface also serves as a valuable piece of documentation, since it tells us at a glance exactly which operations are supported by the type.

Second, an interface provides **data abstraction**. We don't need to know how the interface operations are implemented, and in fact functions coded against the interface will work

with *any* implementation of the interface. For instance, suppose that we'd like to provide a "bounded stacks" data structure, i.e., stacks which don't grow beyond a certain limit. These can be implemented as follows:

```
push (n,xs@[]) x | push (n,xs@(_:_)) x =
  if n>0 then (n-1,x:xs) else throw "full stack";
pop (n,x:xs) = n+1,xs;
top (n,x:xs) = x;
pop (n,[]) = throw "empty stack";
top (n,[]) = throw "empty stack";
```

Note that we represent a bounded stack by a pair `(n,xs)` here, where `xs` is the list of elements and `n` is the "free space" (number of elements we still allow to be pushed). We also add a function to construct such values:

```
bstack n::int xs::list = (n-#xs,xs);
```

Without any further ado, our little RPN calculator works just fine with the new variation of the data structure:

```
> rpn (bstack 3 []) [10,4,3,(+),2,(*),(-)];
2,[-4]
> rpn (bstack 2 []) [10,4,3,(+),2,(*),(-)];
<stdin>, line 7: unhandled exception '"full stack"' while evaluating
'rpn (bstack 2 []) [10,4,3,(+),2,(*),(-)]'
```

While they're quite useful in general, Pure's interface types also have their limitations. In particular, the guarantees provided by an interface are of a purely syntactic nature; the signature doesn't tell us anything about the actual meaning of the provided operations, so unit testing is still needed to ensure certain semantic properties of the implementation. Some further issues due to Pure's dynamically typed nature are discussed under Interfaces in the Caveats and Notes section.

1.6.3 Modules and Imports

Pure doesn't offer separate compilation, but the following type of declaration provides a simple but effective way to assemble a Pure program from several source modules.

```
using_decl  ::=  "using" name ("," name)* ";"
name        ::=  qualified_identifier | string
```

The `using` declaration takes the following form (note that in contrast to symbol declarations, the comma is used as a delimiter symbol here):

```
using name, ...;
```

This causes each given script to be included in the Pure program at the given point (if it wasn't already included before), which makes available all the definitions of the included script in your program. Note that each included script is loaded only *once*, when the first `using` clause for the script is encountered. Nested imports are allowed, i.e., an imported

module may itself import other modules, etc. A Pure program then basically is the concatenation of all the source modules given as command line arguments, with other modules listed in `using` clauses inserted at the corresponding source locations.

(The `using` clause also has an alternative form which allows dynamic libraries and LLVM bitcode modules to be loaded, this will be discussed in the C Interface section.)

For instance, the following declaration causes the math.pure script from the standard library to be included in your program:

```
using math;
```

You can also import multiple scripts in one go:

```
using array, dict, set;
```

Moreover, Pure provides a notation for qualified module names which can be used to denote scripts located in specific package directories, e.g.:

```
using examples::libor::bits;
```

In fact this is equivalent to the following `using` clause which spells out the real filename of the script between double quotes (the `.pure` suffix can also be omitted in which case it is added automatically):

```
using "examples/libor/bits.pure";
```

Both notations can be used interchangeably; the former is usually more convenient, but the latter allows you to denote scripts whose names aren't valid Pure identifiers.

Script identifiers are translated to the corresponding filenames by replacing the '`::`' symbol with the pathname separator '`/`' and tacking on the '`.pure`' suffix. The following table illustrates this with a few examples.

Script identifier	Filename
`math`	`"math.pure"`
`examples::libor::bits`	`"examples/libor/bits.pure"`
`::pure::examples::hello`	`"/pure/examples/hello.pure"`

Note the last example, which shows how an absolute pathname can be denoted using a qualifier starting with '`::`'.

Unless an absolute pathname is given, the interpreter performs a search to locate the script. The search algorithm considers the following directories in the given order:

- the directory of the current script, which is the directory of the script containing the `using` clause, or the current working directory if the clause was read from standard input (as is the case, e.g., in an interactive session);
- the directories named in *`-I`* options on the command line (in the given order);
- the colon-separated list of directories in the `PURE_INCLUDE` environment variable (in the given order);

- finally the directory named by the PURELIB environment variable.

Note that the current working directory is not searched by default (unless the using clause is read from standard input), but of course you can force this by adding the option *-I*. to the command line, or by including '.' in the PURE_INCLUDE variable.

The directory of the current script (the first item above) can be skipped by specifying the script to be loaded as a filename in double quotes, prefixed with the special sys: tag. The search then starts with the "system" directories (*-I*, PURE_INCLUDE and PURELIB) instead. This is useful, e.g., if you want to provide your own custom version of a standard library script which in turn imports that library script. For instance, a custom version of math.pure might employ the following using clause to load the math.pure script from the Pure library:

```
using "sys:math";
// custom definitions go here
log2 x = ln x/ln 2;
```

The interpreter compares script names (to determine whether two scripts are actually the same) by using the *canonicalized* full pathname of the script, following symbolic links to the destination file (albeit only one level). Thus different scripts with the same basename, such as foo/utils.pure and bar/utils.pure can both be included in the same program (unless they link to the same file).

More precisely, canonicalizing a pathname involves the following steps:

- relative pathnames are expanded to absolute ones, using the search rules discussed above;
- the directory part of the pathname is normalized to the form returned by the getcwd system call;
- the ".pure" suffix is added if needed;
- if the resulting script name is actually a symbolic link, the interpreter follows that link to its destination, albeit only one level. (This is only done on Unix-like systems.)

The directory of the canonicalized pathname is also used when searching other scripts included in a script. This makes it possible to have an executable script with a shebang line in its own directory, which is then executed via a symbolic link placed on the system PATH. In this case the script search performed in using clauses will use the real script directory and thus other required scripts can be located there. This is the recommended practice for installing standalone Pure applications in source form which are to be run directly from the shell.

1.6.4 Namespaces

To facilitate modular development, Pure also provides namespaces as a means to avoid name clashes between symbols, and to keep the global namespace tidy and clean. Namespaces serve as containers holding groups of related identifiers and other symbols. Inside each namespace, symbols must be unique, but the same symbol may be used to denote different objects (variables, functions, etc.) in different namespaces. (Pure's namespace system was

heavily inspired by C++ and works in a very similar fashion. So if you know C++ you should feel right at home and skimming this section to pick up Pure's syntax of the namespace constructs should be enough to start using it.)

The global namespace is always available. By default, new symbols are created in this namespace, which is also called the **default namespace**. Additional namespaces can be created with the `namespace` declaration, which also switches to the given namespace (makes it the *current* namespace), so that new symbols are then created in that namespace rather than the default one. The current namespace also applies to all kinds of symbol declarations, including operator and nonfix symbol declarations, as well as `extern` declarations (the latter are described in the C Interface section).

The syntax of namespace declarations is captured by the following grammar rules:

```
namespace_decl  ::=  "namespace" [name] [brackets] ";"
                     | "namespace" name [brackets] "with" item+ "end" ";"
                     | "using" "namespace" [name_spec ("," name_spec)*] ";"
brackets        ::=  "(" left_op right_op ")"
name_spec       ::=  name ["(" qualified_symbol+ ")"]
```

The basic form of the `namespace` declaration looks as follows (there's also a "scoped" form of the `namespace` declaration which will be discussed in Scoped Namespaces at the end of this section):

```
namespace name;
// declarations and definitions in namespace 'name'
namespace;
```

The second form switches back to the default namespace. For instance, in order to define two symbols with the same print name `foo` in two different namespaces `foo` and `bar`, you can write:

```
namespace foo;
foo x = x+1;
namespace bar;
foo x = x-1;
namespace;
```

We can now refer to the symbols we just defined using **qualified symbols** of the form `namespace::symbol`:

```
> foo::foo 99;
100
> bar::foo 99;
98
```

This avoids any potential name clashes, since the qualified identifier notation always makes it clear which namespace the given identifier belongs to.

A namespace can be "reopened" at any time to add new symbols and definitions to it. This allows namespaces to be created that span several source modules. You can also create

several different namespaces in the same module.

Similar to the `using` declaration, a `namespace` declaration accepts either identifiers or double-quoted strings as namespace names. E.g., the following two declarations are equivalent:

```
namespace foo;
namespace "foo";
```

The latter form also allows more descriptive labels which aren't identifiers, e.g.:

```
namespace "Private stuff, keep out!";
```

Note that the namespace prefix in a qualified identifier must be a legal identifier, so it isn't possible to access symbols in namespaces with such descriptive labels in a direct fashion. The only way to get at the symbols in this case is with namespace brackets or by using a `namespace` or `using namespace` declaration (for the latter see Using Namespaces below).

Using Namespaces

Since it is rather inconvenient if you always have to write identifiers in their qualified form outside of their "home" namespace, Pure allows you to specify a list of *search* namespaces which are used to look up symbols not in the default or the current namespace. This is done with the `using namespace` declaration, which takes the following form:

```
using namespace name1, name2, ...;
// ...
using namespace;
```

As with `namespace` declarations, the second form without any namespace arguments gets you back to the default empty list of search namespaces.

For instance, consider this example:

```
namespace foo;
foo x = x+1;
namespace bar;
foo x = x-1;
bar x = x+1;
namespace;
```

The symbols in these namespaces can be accessed unqualified as follows:

```
> using namespace foo;
> foo 99;
100
> using namespace bar;
> foo 99;
98
> bar 99;
100
```

This method is often to be preferred over opening a namespace with the `namespace` declaration, since `using namespace` only gives you "read access" to the imported symbols, so you can't accidentally mess up the definitions of the namespace you're using. Another advantage is that the `using namespace` declaration also lets you search multiple namespaces at once:

```
using namespace foo, bar;
```

Be warned, however, that this brings up the very same issue of name clashes again:

```
> using namespace foo, bar;
> foo 99;
<stdin>, line 15: symbol 'foo' is ambiguous here
```

In such a case you'll have to resort to using namespace qualifiers again, in order to resolve the name clash:

```
> foo::foo 99;
100
```

To avoid this kind of mishap, you can also selectively import just a few symbols from a namespace instead. This can be done with a declaration of the following form:

```
using namespace name1 ( sym1 sym2 ... ), name2 ... ;
```

As indicated, the symbols to be imported can optionally be placed as a whitespace-delimited list inside parentheses, following the corresponding namespace name. (As with symbol declarations, the symbols may optionally be qualified with a namespace prefix, which must match the imported namespace here.) For instance:

```
> using namespace foo, bar (bar);
> foo 99;
100
> bar 99;
100
> bar::foo 99;
98
```

Note that now we have no clash on the `foo` symbol any more, because we restricted the import from the `bar` namespace to the `bar` symbol, so that `bar::foo` has to be denoted with a qualified symbol now.

Symbol Lookup and Creation

Pure's rules for looking up and creating symbols are fairly straightforward and akin to those in other languages featuring namespaces. However, there are some intricacies involved, because the rewriting rule format of definitions allows "referential" use of symbols not only in the "body" (right-hand side) of a definition, but also in the left-hand side patterns. We discuss this in detail below.

The compiler searches for symbols first in the current namespace (if any), then in the currently active search namespaces (if any), and finally in the default (i.e., the global) namespace, in that order. This automatic lookup can be bypassed by using an *absolute* namespace qualifier of the form `::foo::bar`. In particular, `::bar` always denotes the symbol `bar` in the default namespace, while `::foo::bar` denotes the symbol `bar` in the `foo` namespace. (Normally, the latter kind of notation is only needed if you have to deal with nested namespaces, see Hierarchical Namespaces below.)

If no existing symbol is found, a new symbol is created automatically, by implicitly declaring a public symbol with default attributes. New *unqualified* symbols are always created in the current namespace, while new *qualified* symbols are created in the namespace given by the namespace prefix of the symbol.

Note: Pure's implicit symbol declarations are a mixed blessing. They are convenient, especially in interactive usage, but they also let missing or mistyped symbols go unnoticed much too easily. As a remedy, in the case of qualified symbols the compiler checks that the given namespace prefix matches the current namespace, in order to catch typos and other silly mistakes and prevent you from accidentally clobbering the contents of other namespaces. For instance:

```
> namespace foo;
> namespace;
> foo::bar x = 1/x;
<stdin>, line 3: undeclared symbol 'foo::bar'
```

To make these errors go away it's enough to just declare the symbols in their proper namespaces.

In addition, you can run the interpreter with the `-w` option (see Invoking Pure) to check your scripts for (non-defining) uses of undeclared unqualified function symbols. This is highly recommended. For instance, in the following example we forgot to import the `system` module which defines the `puts` function. Running the interpreter with `-w` highlights such potential errors:

```
$ pure -w
> puts "bla"; // missing import of system module
<stdin>, line 1: warning: implicit declaration of 'puts'
puts "bla"
```

For legitimate uses (such as forward uses of a symbol which is defined later), you can make these warnings go away by declaring the symbol before using it.

New symbols are also created if a global unqualified (and yet undeclared) symbol is being "defined" in a rewriting rule or `let`/`const` definition, even if a symbol with the same print name from another namespace is already visible in the current scope. To distinguish "defining" from "referring" uses of a global symbol, Pure uses the following (purely syntactic) notions:

- A **defining occurrence** of a global *function*, *macro* or *type symbol* is any occurrence of the

symbol as the (leftmost) *head symbol* on the left-hand side of a rewriting rule.

- A **defining occurrence** of a global *variable* or *constant symbol* is any occurrence of the symbol in a *variable position* (as given by the "head = function" rule, cf. Variables in Equations) on the left-hand side of a `let` or `const` definition.
- All other occurrences of global symbols on the left-hand side, as well as *all* symbol occurrences on the right-hand side of a definition are **referring occurrences**. (Note that this also subsumes all occurrences of *type tags* on the left-hand side of an equation.)

The following example illustrates these notions:

```
namespace foo;
bar (bar x) = bar x;
let x,y = 1,2;
namespace;
```

Here, the first occurrence of `bar` on the left-hand side `bar (bar x)` of the first rule is a *defining* occurrence, as are the occurrences of `x` and `y` on the left-hand side of the `let` definition. Hence these symbols are created as new symbols in the namespace `foo`. On the other hand, the other occurrences of `bar` in the first rule, as well as the `,` symbol on the left-hand side of the `let` definition are *referring* occurrences. In the former case, `bar` refers to the `bar` symbol defined by the rule, while in the latter case the `,` operator is actually declared in the prelude and thus imported from the global namespace.

The same rules of lookup also apply to type tags on the left-hand side of an equation, but in this case the interpreter will look specifically for type symbols, avoiding any other kinds of symbols which might be visible in the same context. Thus, in the following example, the type tag `bar` is correctly resolved to `bar::bar`, even though the (function) symbol `foo::bar` is visible at this point:

```
> namespace bar;
> type bar;
> namespace foo;
> public bar;
> using namespace bar;
> foo x::bar = bar x;
> show foo::foo
foo::foo x :: bar::bar = foo::bar x;
```

Note that special operator (and nonfix) symbols *always* require an explicit declaration. This works as already discussed in the Symbol Declarations section, except that you first switch to the appropriate namespace before declaring the symbols. For instance, here is how you can create a new + operation which multiplies its operands rather than adding them:

```
> namespace my;
> infixl 2200 +;
> x+y = x*y;
> 5+7;
35
```

Note that the new + operation really belongs to the namespace we created. The + operation

in the default namespace works as before, and in fact you can use qualified symbols to pick the version that you need:

```
> namespace;
> 5+7;
12
> 5 ::+ 7;
12
> 5 my::+ 7;
35
```

Here's what you get if you happen to forget the declaration of the + operator:

```
> namespace my;
> x+y = x*y;
<stdin>, line 2: infixl symbol '+' was not declared in this namespace
```

Thus the compiler will never create a new instance of an operator symbol on the fly, an explicit declaration is always needed in such cases.

Note that if you *really* wanted to redefine the global + operator, you can do this even while the `my` namespace is current. You just have to use a qualified identifier in this case, as follows:

```
> namespace my;
> x ::+ y = x*y;
> a+b;
a*b
```

This should rarely be necessary (in the above example you might just as well enter this rule while in the global namespace), but it can be useful in some circumstances. Specifically, you might want to "overload" a global function or operator with a definition that makes use of private symbols of a namespace (which are only visible inside that namespace; see Private Symbols below). For instance:

```
> namespace my;
> private bar;
> bar x y = x*y;
> x ::+ y = bar x y;
> a+b;
a*b
```

(The above is a rather contrived example, since the very same functionality can be accomplished much easier, but there are some situations where this method is needed.)

Private Symbols

Pure also allows you to have private symbols, as a means to hide away internal operations which shouldn't be accessed directly outside the namespace in which they are declared. The scope of a private symbol is confined to its namespace, i.e., the symbol is only visible when its "home" namespace is current. Symbols are declared private by using the `private` keyword in the symbol declaration:

```
> namespace secret;
> private baz;
> // 'baz' is a private symbol in namespace 'secret' here
> baz x = 2*x;
> // you can use 'baz' just like any other symbol here
> baz 99;
198
> namespace;
```

Note that, at this point, `secret::baz` is now invisible, even if you have `secret` in the search namespace list:

```
> using namespace secret;
> // this actually creates a 'baz' symbol in the default namespace:
> baz 99;
baz 99
> secret::baz 99;
<stdin>, line 27: symbol 'secret::baz' is private here
```

The only way to bring the symbol back into scope is to make the `secret` namespace current again:

```
> namespace secret;
> baz 99;
198
> secret::baz 99;
198
```

Namespace Brackets

All the namespace-related constructs we discussed so far only provide a means to switch namespaces on a per-rule basis. Sometimes it is convenient if you can switch namespaces on the fly inside an expression. This is especially useful if you want to embed a domain-specific sublanguage (DSL) in Pure. DSLs typically provide their own system of operators which differ from the standard Pure operators and thus need to be declared in their own namespace.

To make this possible, Pure allows you to associate a namespace with a corresponding pair of outfix symbols. This turns the outfix symbols into special **namespace brackets** which can then be used to quickly switch namespaces in an expression by just enclosing a subexpression in the namespace brackets.

To these ends, the syntax of `namespace` declarations allows you to optionally specify a pair of outfix symbols inside parentheses after the namespace name. The outfix symbols to be used as namespace brackets must have been declared beforehand. For instance:

```
outfix « »;
namespace foo (« »);
infixr (::^) ^;
```

```
x^y = 2*x+y;
namespace;
```

The code above introduces a `foo` namespace which defines a special variation of the `(^)` operator. It also associates the namespace with the « » brackets so that you can write:

```
> (a+b)^c+10;
(a+b)^c+10
> «(a+b)^c»+10;
2*(a+b)+c+10
```

Note the use of the namespace brackets in the second input line. This changes the meaning of the `^` operator, which now refers to `foo::^` instead. Also note that the namespace brackets themselves are removed from the resulting expression; they are only used to temporarily switch the namespace to `foo` inside the bracketed subexpression. This works pretty much like a `namespace` declaration (so any active search namespaces remain in effect), but is limited in scope to the bracketed subexpression and only gives access to the public symbols of the namespace (like a `using namespace` declaration would do).

The rules of visibility for the namespace bracket symbols themselves are the same as for any other symbols. So they need to be in scope if you want to denote them in unqualified form (which is always the case if they are declared in the default namespace, as in the example above). If necessary, you can also specify them in their qualified form as usual.

Namespace brackets can be used anywhere inside an expression, even on the left-hand side of a rule. So, for instance, we might also have written the example above as follows:

```
outfix « »;
namespace foo (« »);
infixr (::^) ^;
namespace;

«x^y» = 2*x+y;
```

Note the use of the namespace brackets on the last line. This rule actually expands to:

```
x foo::^ y  = 2*x+y;
```

The special meaning of namespace brackets can be turned off and back on again at any time with a corresponding `namespace` declaration. For instance:

```
> namespace (« »); // turn off the special meaning of « »
> «(a+b)^c»+10;
« (a+b)^c »+10
> namespace foo (« »); // turn it on again
> namespace;
> «(a+b)^c»+10;
2*(a+b)+c+10
```

(Note that as a side effect these declarations also change the current namespace, so that we use the `namespace;` declaration in the second last line to change back to the default namespace.)

As shown in the first line of the example above, a namespace brackets declaration without a namespace just turns off the special processing of the brackets. In order to define a namespace bracket for the *default* namespace, you need to explicitly specify an empty namespace instead, as follows:

```
> outfix «: :»;
> namespace "" («: :»);
> «(a+b)^«:x^y:»»;
2*(a+b)+x^y
```

As this example illustrates, namespace brackets can also be nested, which is useful, e.g., if you need to combine subexpressions from several DSLs in a single expression. In this example we employ the «:x^y:» subexpression to temporarily switch back to the default namespace inside the « »-bracketed expression which is parsed in the `foo` namespace.

Hierarchical Namespaces

Namespace identifiers can themselves be qualified identifiers in Pure, which enables you to introduce a hierarchy of namespaces. This is useful, e.g., to group related namespaces together under a common "umbrella" namespace:

```
namespace my;
namespace my::old;
foo x = x+1;
namespace my::new;
foo x = x-1;
```

Note that the namespace `my`, which serves as the parent namespace, must be created before the `my::old` and `my::new` namespaces, even if it does not contain any symbols of its own. After these declarations, the `my::old` and `my::new` namespaces are part of the `my` namespace and will be considered in name lookup accordingly, so that you can write:

```
> using namespace my;
> old::foo 99;
100
> new::foo 99;
98
```

This works pretty much like a hierarchy of directories and files, where the namespaces play the role of the directories (with the default namespace as the root directory), the symbols in each namespace correspond to the files in a directory, and the `using namespace` declaration functions similar to the shell's `PATH` variable.

Sometimes it is necessary to tell the compiler to use a symbol in a specific namespace, bypassing the usual symbol lookup mechanism. For instance, suppose that we introduce another *global* `old` namespace and define yet another version of `foo` in that namespace:

```
namespace old;
foo x = 2*x;
namespace;
```

Now, if we want to access that function, with `my` still active as the search namespace, we cannot simply refer to the new function as `old::foo`, since this name will resolve to `my::old::foo` instead. As a remedy, the compiler accepts an **absolute** qualified identifier of the form `::old::foo`. This bypasses name lookup and thus always yields exactly the symbol in the given namespace (if it exists; as mentioned previously, the compiler will complain about an undeclared symbol otherwise):

```
> old::foo 99;
100
> ::old::foo 99;
198
```

Also note that, as a special case of the absolute qualifier notation, `::foo` always denotes the symbol `foo` in the default namespace.

Scoped Namespaces

Pure also provides an alternative scoped `namespace` construct which makes nested namespace definitions more convenient. This construct takes the following form:

```
namespace name with ... end;
```

The part between `with` and `end` may contain arbitrary declarations and definitions, using the same syntax as the toplevel. These are processed in the context of the given namespace, as if you had written:

```
namespace name;
...
namespace;
```

However, the scoped namespace construct always returns you to the namespace which was active before, and thus these declarations may be nested:

```
namespace foo with
  // declarations and definitions in namespace foo
  namespace bar with
    // declarations and definitions in namespace bar
  end;
  // more declarations and definitions in namespace foo
end;
```

Note that this kind of nesting does not necessarily imply a namespace hierarchy as discussed in Hierarchical Namespaces. However, you can achieve this by using the appropriate qualified namespace names:

```
namespace foo with
  // ...
  namespace foo::bar with
    // ...
  end;
```

```
  // ...
end;
```

Another special feature of the scoped namespace construct is that `using namespace` declarations are always local to the current namespace scope (and other nested namespace scopes inside it). Thus the previous setting is restored at the end of each scope:

```
using namespace foo;
namespace foo with
  // still using namespace foo here
  using namespace bar;
  // now using namespace bar
  namespace bar with
    // still using namespace bar here
    using namespace foo;
    // now using namespace foo
  end;
  // back to using namespace bar
end;
// back to using namespace foo at toplevel
```

Finally, here's a more concrete example which shows how scoped namespaces might be used to declare two namespaces and populate them with various functions and operators:

```
namespace foo with
  infixr (::^) ^;
  foo x = x+1;
  bar x = x-1;
  x^y = 2*x+y;
end;

namespace bar with
  outfix <: :>;
  foo x = x+2;
  bar x = x-2;
end;

using namespace foo(^ foo), bar(bar <: :>);

// namespace foo
foo x;
x^y;

// namespace bar
bar x;
<: x,y :>;
```

Pure's namespaces can thus be used pretty much like "packages" or "modules" in languages like Ada or Modula-2. They provide a structured way to describe program components offering collections of related data and operations, which can be brought into scope in a controlled way by making judicious use of `using namespace` declarations. They also provide an abstraction barrier, since internal operations and data structures can be hidden away

employing private symbols.

Please note that these facilities are not Pure's main focus and thus they are somewhat limited compared to programming languages specifically designed for big projects and large teams of developers. Nevertheless they should be useful if your programs grow beyond a small collection of simple source modules, and enable you to manage most Pure projects with ease.

1.7 Macros

Macros are a special type of functions to be executed as a kind of "preprocessing stage" at compile time. In Pure these are typically used to define custom special forms and to perform inlining of function calls and other kinds of source-level optimizations.

Whereas the macro facilities of most programming languages simply provide a kind of textual substitution mechanism, Pure macros operate on symbolic expressions and are implemented by the same kind of rewriting rules that are also used to define ordinary functions in Pure. This makes them robust and easy to use for most common preprocessing purposes.

Syntactically, a macro definition looks just like a function definition with the `def` keyword in front of it. Only unconditional rewriting rules are permitted here, i.e., rules without guards and multiple right-hand sides. However, multiple left-hand sides can be employed as usual to abbreviate a collection of rules with the same left-hand side, as described in the General Rules section.

The major difference between function and macro definitions is that the latter are processed at compile time rather than run time. To these ends, macro calls on the right-hand sides of function, constant and variable definitions are evaluated by reducing them to normal form using the available macro rules. The resulting expressions are then substituted for the macro calls. All macro substitution happens before constant substitutions and the actual compilation step. Macros can be defined in terms of other macros (also recursively), and are normally evaluated using call by value (i.e., macro calls in macro arguments are expanded before the macro gets applied to its parameters).

In the first half of this section we start out with some common uses of macros which should cover most aspects of macro programming that the average Pure programmer will need. The remainder of this section then discusses some more advanced features of Pure's macro system intended for power users.

1.7.1 Optimization Rules

Let's begin with a simple example of an optimization rule from the prelude, which eliminates saturated instances of the right-associative function application operator (you can find this near the beginning of prelude.pure):

```
def f $ x = f x;
```

Like in Haskell, '$' in fact just denotes function application, but it is a low-priority operator which is handy to write cascading function calls. With the above macro rule, these will be "inlined" as ordinary function applications automatically. Example:

```
> foo x = bar $ bar $ 2*x;
> show foo
foo x = bar (bar (2*x));
```

Note that a macro may have the same name as an ordinary Pure function, which is essential if you want to inline calls to an existing function. (Just like ordinary functions, the number of parameters in each rule for a given macro must be the same, but a macro may have a different number of arguments than the corresponding function.)

When running interactively, you can follow the reduction steps the compiler performs during macro evaluation. To these ends, you have to set "tracepoints" on the relevant macros, using the `trace` command with the `-m` option; see Interactive Commands. (This works even if the interpreter is run in non-debugging mode.) Note that since macro expansion is performed at compile time, you'll have to do this *before* entering the definitions in which the macro is used. However, in many cases you can also just enter the right-hand side of the equation at the interpreter prompt to see how it gets expanded. For instance:

```
> trace -m $
> bar $ bar $ 2*x;
-- macro ($): bar$2*x --> bar (2*x)
-- macro ($): bar$bar (2*x) --> bar (bar (2*x))
bar (bar (2*x))
```

Now let's see how we can add our own optimization rules. Suppose we'd like to expand saturated calls of the `succ` function. This function is defined in the prelude; it just adds 1 to its single argument. We can inline such calls as follows:

```
> def succ (x+y) = x+(y+1);
> def succ x = x+1;
> foo x = succ (succ (succ x));
> show foo
foo x = x+3;
```

Again, let's see exactly what's going on there:

```
> trace -m succ
> succ (succ (succ x));
-- macro succ: succ x --> x+1
-- macro succ: succ (x+1) --> x+(1+1)
-- macro succ: succ (x+(1+1)) --> x+(1+1+1)
x+3
```

Note that the contraction of the subterm `1+1+1` to the integer constant `3` is actually done by the compiler after macro expansion has been performed. This is also called "constant folding", see Constant Definitions in the Caveats and Notes section for details. It is also the reason that we added the first rule for `succ`. This rule may seem superflous at first sight, but actually it is needed to massage the sum into a form which enables constant folding.

Rules like these can help the compiler generate better code. Of course, the above examples are still rather elementary. Pure macros can do much more elaborate optimizations, but for this we first need to discuss how to write recursive macros, as well as macros which take apart special terms like lambdas. After that we'll return to the subject of optimization rules in Advanced Optimization below.

1.7.2 Recursive Macros

Macros can also be recursive, in which case they usually consist of multiple rules and make use of pattern-matching just like ordinary function definitions.

Note: Pure macros are just as powerful as (unconditional) term rewriting systems and thus they are Turing-complete. This implies that a badly written macro may well send the Pure compiler into an infinite recursion, which results in a stack overflow at compile time. See Stack Size and Tail Recursion in the Caveats and Notes section for information on how to deal with this by setting the `PURE_STACK` environment variable.

As a simple example, let's see how we can inline invocations of the `#` size operator on list constants:

```
def #[] = 0;
def #(x:xs) = #xs+1;
```

As you can see, the definition is pretty straightforward; exactly the same rules might also be used for an ordinary function definition, although the standard library actually implements `#` a bit differently to make good use of tail recursion. Let's check that this actually works:

```
> foo = #[1,2,3,4];
> show foo
foo = 4;
```

Note that the result of macro expansion is actually `0+1+1+1+1` here, you can check that by running the macro with `trace -m #`. Constant folding contracts this to `4` after macro expansion, as explained in the previous subsection.

This was rather easy. So let's implement a more elaborate example: a basic Pure version of Lisp's quasiquote which allows you to create a quoted expression from a "template" while substituting variable parts of the template. (For the sake of brevity, we present a somewhat abridged version here which does not cover all corner cases. The full version of this macro can be found as lib/quasiquote.pure in the Pure distribution.)

```
def quasiquote (unquote x)      = x;
def quasiquote (f@_ (splice x)) = foldl ($) (quasiquote f) x;
def quasiquote (f@_ x)          = quasiquote f (quasiquote x);
def quasiquote x                = quote x;
```

(Note the `f@_`, which is an anonymous "as" pattern forcing the compiler to recognize `f` as a function variable, rather than a literal function symbol. See "As" Patterns in the Caveats

and Notes section for an explanation of this trick.)

The first rule above takes care of "unquoting" embedded subterms. The second rule "splices" an argument list into an enclosing function application. The third rule recurses into subterms of a function application, and the fourth and last rule takes care of quoting the "atomic" subterms. Note that `unquote` and `splice` themselves are just passive constructor symbols, the real work is done by `quasiquote`, using `foldl` at runtime to actually perform the splicing. (Putting off the splicing until runtime makes it possible to splice argument lists computed at runtime.)

If we want, we can also add some syntactic sugar for Lisp weenies. (Note that we cannot have `‘,‘` for unquoting, so we use `‘,$‘` instead.)

```
prefix 9 ‘ ,$ ,@ ;
def ‘x = quasiquote x; def ,$x = unquote x; def ,@x = splice x;
```

Examples:

```
> ‘(2*42+2^12);
2*42+2^12
> ‘(2*42+,$(2^12));
2*42+4096.0
> ‘foo 1 2 (,@’[2/3,3/4]) (5/6);
foo 1 2 (2/3) (3/4) (5/6)
> ‘foo 1 2 (,@args) (5/6) when args = ’[2/3,3/4] end;
foo 1 2 (2/3) (3/4) (5/6)
```

1.7.3 User-Defined Special Forms

The `quasiquote` macro in the preceding subsection also provides an example of how you can use macros to define your own special forms. This works because the actual evaluation of macro arguments is put off until runtime, and thus we can safely pass them to built-in special forms and other constructs which defer their evaluation *at runtime*. In fact, the right-hand side of a macro rule may be an arbitrary Pure expression involving conditional expressions, lambdas, binding clauses, etc. These are never evaluated during macro substitution, they just become part of the macro expansion (after substituting the macro parameters).

Here is another useful example of a user-defined special form, the macro `timex` which employs the system function `clock` to report the cpu time in seconds needed to evaluate a given expression, along with the computed result:

```
> using system;
> def timex x = (clock-t0)/CLOCKS_PER_SEC,y when t0 = clock; y = x end;
> sum = foldl (+) 0L;
> timex $ sum (1L..100000L);
0.43,5000050000L
```

Note that the above definition of `timex` wouldn't work as an ordinary function definition, since by virtue of Pure's basic eager evaluation strategy the `x` parameter would have been

evaluated already before it is passed to `timex`, making `timex` always return a zero time value. Try it!

1.7.4 Macro Hygiene

Pure macros are lexically scoped, i.e., the binding of symbols in the right-hand-side of a macro definition is determined statically by the text of the definition, and macro parameter substitution also takes into account binding constructs, such as `with` and `when` clauses, in the right-hand side of the definition. Macro facilities with these pleasant properties are also known as **hygienic macros**. They are not susceptible to so-called "name capture," which makes macros in less sophisticated languages bug-ridden and hard to use.

Macro hygiene is a somewhat esoteric topic for most programmers, so let us take a brief look at what it's all about. The problem avoided by hygienic macros is that of *name capture*. There are actually two kinds of name capture which may occur in unhygienic macro systems:

- A free symbol in the macro *body* inadvertently becomes bound to the value of a local symbol in the context in which the macro is called.
- A free symbol in the macro *call* inadvertently becomes bound to the value of a local symbol in the macro body.

Pure's hygienic macros avoid both pitfalls. Here is an example for the first form of name capture:

```
> def G x = x+y;
> G 10 when y = 99 end;
10+y
```

Note that the expansion of the `G` macro correctly uses the global instance of `y`, even though `y` is locally defined in the context of the macro call. (In some languages this form of name capture is sometimes used deliberately in order to make the macro use the binding of the symbol which is active at the point of the macro call. Normally, this won't work in Pure, although there is a way to force this behaviour in Pure as well, see Name Capture in the Caveats and Notes section.)

In contrast, the second form of name capture is usually not intended, and is therefore more dangerous. Consider the following example:

```
> def F x = x+y when y = x+1 end;
> F y;
y+(y+1)
```

Pure again gives the correct result here. You'd have to be worried if you got `(y+1)+(y+1)` instead, which would result from the literal expansion `y+y when y = y+1 end`, where the (free) variable `y` passed to `F` gets captured by the local binding of `y`. In fact, that's exactly what you get with C macros:

```
#define F(x) { int y = x+1; return x+y; }
```

Here `F(y)` expands to `{ int y = y+1; return y+y; }` which is usually *not* what you want.

This completes our little introduction to Pure's macro facilities. The above material should in fact cover all the common uses of macros in Pure. However, if you want to become a real Pure macro wizard then read on. In the following subsections we're going to discover some more advanced features of Pure's macro system which let you write macros for manipulating special forms and give you access to Pure's reflection capabilities.

1.7.5 Built-in Macros and Special Expressions

As already mentioned in The Quote, special expressions such as conditionals and lambdas cannot be directly represented as runtime data in Pure. But they can be *quoted* in which case they are replaced by corresponding "placeholder terms". These placeholder terms are in fact implemented as built-in macros which, when evaluated, construct the corresponding specials.

macro **__ifelse__** x y z
: This macro expands to the conditional expression `if x then y else z` during macro evaluation.

macro **__lambda__** [x1,...,xn] y
: Expands to the lambda expression `\x1 ... xn -> y`.

macro **__case__** x [(x1 –> y1),...,(xn –> yn)]
: Expands to the `case` expression `case x of x1 = y1; ...; xn = yn end`. Note that the `-->` symbol is used to separate the left-hand side and the right-hand side of each rule (see below).

macro x **__when__** [(x1 –> y1),...,(xn –> yn)]
: Expands to the `when` expression `x when x1 = y1; ...; xn = yn end`. Here the left-hand side of a rule may be omitted if it is just the anonymous variable; i.e., `x __when__ [foo y]` is the same as `x __when__ [_ --> foo y]`.

macro x **__with__** [(x1 –> y1),...,(xn –> yn)]
: Expands to the `with` expression `x with x1 = y1; ...; xn = yn end`.

Note that the following low-priority infix operators are used to denote equations in the `__case__`, `__when__` and `__with__` macros:

constructor x **-->** y
: Denotes an equation `x = y`.

constructor x **__if__** y
: Attaches a guard to the right-hand side of an equation. That is, `x --> y __if__ z` denotes the conditional equation `x = y if z`. This symbol is only recognized in `__case__` and `__with__` calls.

In addition, patterns on the left-hand side of equations or in lambda arguments may be decorated with the following constructor terms to indicate "as" patterns and type tags (these are infix operators with a very high priority):

constructor x **__as__** y
: Denotes an "as" pattern `x @ y`.

constructor x **__type__** y
: Denotes a type tag `x :: y`.

Note that all these symbols are in fact just constructors which are only interpreted in the context of the built-in macros listed above; they aren't macros themselves.

It's good to remember the above when you're doing macro programming. However, to see the placeholder term of a special, you can also just type a quoted expression in the interpreter:

```
> '(\x->x+1);
__lambda__ [x] (x+1)
> '(f with f x = y when y = x+1 end end);
f __with__ [f x-->y __when__ [y-->x+1]]
```

List and matrix comprehensions can also be quoted. These are basically syntactic sugar for lambda applications, cf. Primary Expressions. The compiler expands them to their "unsugared" form already before macro substitution, so no special kinds of built-in macros are needed to represent them. When quoted, comprehensions are thus denoted in their unsugared form, which consists of a pile of lambda expressions and list or matrix construction functions for the generation clauses, and possibly some conditionals for the filter clauses of the comprehension. For instance:

```
> '[2*x | x = 1..3];
listmap (__lambda__ [x] (2*x)) (1..3)
```

Here's how type tags and "as" patterns in quoted specials look like:

```
> '(\x::int->x+1);
__lambda__ [x __type__ int] (x+1)
> '(dup (1..3) with dup xs@(x:_) = x:xs end);
dup (1..3) __with__ [dup (xs __as__ (x:_))-->x:xs]
```

Note that the placeholder terms for the specials are quoted here, and hence they are not evaluated (quoting inhibits macro expansion, just like it prevents the evaluation of ordinary function calls). Evaluating the placeholder terms executes the corresponding specials:

```
> '(dup (1..3) with dup xs@(x:_) = x:xs end);
dup (1..3) __with__ [dup (xs __as__ (x:_))-->x:xs]
> eval ans;
[1,1,2,3]
```

Of course, you can also just enter the macros directly (without quoting) to have them evaluated:

```
> dup (1..3) __with__ [dup (xs __as__ (x:_))-->x:xs];
[1,1,2,3]
> __lambda__ [x __type__ int] (x+1);
#<closure 0x7f1934158dc8>
```

```
> ans 99;
100
```

The `__str__` function can be used to pretty-print quoted specials:

```
> __str__ ('__lambda__ [x __type__ int] (x+1));
"\\x::int -> x+1"
> __str__ ('(dup (1..3) __with__ [dup (xs __as__ (x:_))-->x:xs]));
"dup (1..3) with dup xs@(x:_) = x:xs end"
```

This is useful to see which expression a quoted special will expand to. Note that `__str__` can also be used to define print representations for quoted specials with `__show__` (described in Pretty-Printing) if you always want to have them printed that way by the interpreter.

As quoted specials are just ordinary Pure expressions, they can be manipulated by functions just like any other term. For instance, here's how you can define a function which takes a quoted lambda and swaps its two arguments:

```
> swap (__lambda__ [x,y] z) = '(__lambda__ [y,x] z);
> swap ('(\a b->a-b));
__lambda__ [b,a] (a-b)
> eval ans 2 3; // same as (\b a->a-b) 2 3
1
```

For convenience, a literal special expression can also be used on the left-hand side of an equation, in which case it actually denotes the corresponding placeholder term. So the `swap` function can also be defined like this (note that we first scratch the previous definition of `swap` with the `clear` command, see Interactive Commands):

```
> clear swap
> swap (\x y -> z) = '(\y x -> z);
> swap ('(\a b->a-b));
__lambda__ [b,a] (a-b)
```

This is usually easier to write and improves readability. However, there are cases in which you want to work with the built-in macros in a direct fashion. In particular, this becomes necessary when writing more generic rules which deal, e.g., with lambdas involving a variable number of arguments, or if you need real (i.e., unquoted) type tags or "as" patterns in a placeholder pattern. We'll see examples of these later.

Quoted specials can be manipulated with macros just as well as with functions. In fact, this is quite common and thus the macro evaluator has some special support to make this more convenient. Specifically, it is possible to make a macro quote its arguments in an automatic fashion, by means of the `--quoteargs` pragma. To illustrate this, let's redefine `swap` as a macro:

```
> clear swap
> #! --quoteargs swap
> def swap (\x y -> z) = '(\y x -> z);
> swap (\a b->a-b);
__lambda__ [b,a] (a-b)
```

The *--quoteargs* pragma makes the swap macro receive its argument unevaluated, as if it was quoted (but without a literal quote around it). Therefore the quote on the lambda argument of swap can now be omitted. However, the result is still a quoted lambda. It's tempting to just omit the quote on the right-hand side of the macro definition as well, in order to get a real lambda instead:

```
> clear swap
> def swap (\x y -> z) = \y x -> z;
> swap (\a b->a-b);
#<closure 0x7f1934156f00>
> ans 2 3;
a-b
```

We got a closure all right, but apparently it's not the right one. Let's use `trace -m` to figure out what went wrong:

```
> trace -m swap
> swap (\a b->a-b);
-- macro swap: swap (\a b -> a-b) --> \y x -> a-b
#<closure 0x7f1934157248>
```

Ok, so the result is the lambda `\y x -> a-b`, not `\b a -> a-b` as we expected. This happens because we used a literal (unquoted) lambda on the right-hand side, which does its own variable binding; consequently, the variables x and y are bound by the lambda in this context, not by the left-hand side of the macro rule.

So just putting an unquoted lambda on the right-hand side doesn't do the job. One way to deal with the situation is to just employ the `__lambda__` macro in a direct way, as we've seen before:

```
> clear swap
> def swap (__lambda__ [x,y] z) = __lambda__ [y,x] z;
> swap (\a b->a-b);
-- macro swap: swap (\a b -> a-b) --> __lambda__ [b,a] (a-b)
-- macro __lambda__: __lambda__ [b,a] (a-b) --> \b a -> a-b
#<closure 0x7f1934156f00>
> ans 2 3;
1
```

This works, but doesn't look very nice. Often it's more convenient to first construct a quoted term involving the necessary specials and then have it evaluated during macro evaluation. Pure provides yet another built-in macro for this purpose:

macro __eval__ x

Evaluate x at macro expansion time. This works by stripping one level of (outermost) quotes from x and performing macro expansion on the resulting unquoted subexpressions.

Using `__eval__`, we can implement the swap macro as follows:

```
> clear swap
> def swap (\x y -> z) = __eval__ ('(\y x -> z));
> swap (\a b->a-b);
```

```
-- macro swap: swap (\a b -> a-b) --> __eval__ ('__lambda__ [b,a] (a-b))
-- macro __lambda__: __lambda__ [b,a] (a-b) --> \b a -> a-b
-- macro __eval__: __eval__ ('__lambda__ [b,a] (a-b)) --> \b a -> a-b
#<closure 0x7f7e1f867dc8>
> ans 2 3;
1
```

Lisp programmers should note the difference. In Lisp, macros usually yield a quoted expression which is evaluated implicitly during macro expansion. This is never done automatically in Pure, since many Pure macros work perfectly well without it. Instead, quotes in a macro expansion are treated as literals, and you'll have to explicitly call `__eval__` to remove them during macro evaluation.

A final caveat: Placeholder terms for specials are just simple expressions; they don't do any variable binding by themselves. Thus the rules of macro hygiene don't apply to them, which makes it possible to manipulate lambdas and local definitions in any desired way. On the other hand, this means that it is the programmer's responsibility to avoid accidental name capture when using these facilities. Most macro code will work all right when written in a straightforward way, but there are some corner cases which need special attention (cf. Name Capture).

Sometimes the only convenient way to avoid name capture is to create new symbols on the fly. This will often be necessary if a macro generates an entire block construct (`case`, `when`, `with` or lambda) from scratch. The following built-in macro is provided for this purpose:

macro __gensym__

Create a new unqualified symbol which is guaranteed to not exist at the time of the macro call. These symbols typically look like `__x123__` and can be used for any purpose (i.e., as global or local as well as function or variable symbols).

For instance, here's how we can implement a macro `foo` which creates a lambda from a given argument, using `__gensym__` to generate a fresh local variable for the lambda argument. This guarantees that variables in the argument expression don't get captured by the lambda variable when the closure is created with a call to the built-in `__lambda__` macro.

```
> def foo x = bar __gensym__ x;
> def bar x y = __lambda__ [x] (x+y);
> trace -m foo
> foo (a*b);
-- macro foo: foo (a*b) --> bar __gensym__ (a*b)
-- macro __gensym__: __gensym__ --> __x1__
-- macro bar: bar __x1__ (a*b) --> __lambda__ [__x1__] (__x1__+a*b)
-- macro __lambda__: __lambda__ [__x1__] (__x1__+a*b) --> \__x1__ -> __x1__+a*b
#<closure 0x7f66f6c88db0>
> ans 77;
77+a*b
```

The `__gensym__` macro returns a new variable for each invocation, and always ensures that it doesn't accidentally reuse a symbol already introduced by the user (even if it looks like a symbol that `__gensym__` might itself create):

```
> foo (a*__x2__);
-- macro foo: foo (a*__x2__) --> bar __gensym__ (a*__x2__)
-- macro __gensym__: __gensym__ --> __x3__
-- macro bar: bar __x3__ (a*__x2__) --> __lambda__ [__x3__] (__x3__+a*__x2__)
-- macro __lambda__: __lambda__ [__x3__] (__x3__+a*__x2__) --> \__x3__ -> __x3__+a*__x2__
#<closure 0x7f66f6c887e8>
> ans 77;
77+a*__x2__
```

1.7.6 Advanced Optimization

We are now in a position to have a look at some of the trickier optimization macros defined in the prelude. The following `__do__` macro can be found near the end of the prelude.pure module; it is used to optimize the case of "throwaway" list and matrix comprehensions. This is useful if a comprehension is evaluated solely for its side effects. To keep things simple, we discuss a slightly abridged version of the `__do__` macro which only deals with list comprehensions and ignores some obscure corner cases. You can find this version in the examples/do.pure script. Please also check the prelude for the full version of this macro.

Note that we define our own versions of `void` and `__do__` here which are placed into the `my` namespace to avoid conflicts with the prelude.

```
namespace my;

void _ = ();

#! --quoteargs my::__do__

def void [x] = void x;
def void (catmap f x) | void (listmap f x) = __do__ f x;

// Recurse into embedded generator clauses.
def __do__ (__lambda__ [x] y@(listmap _ _)) |
    __do__ (__lambda__ [x] y@(catmap _ _)) =
    __do__ $ (__lambda__ [x] (void y));

// Recurse into embedded filter clauses.
def __do__ (__lambda__ [x] (__ifelse__ y z [])) =
    __do__ $ (__lambda__ [x] (__ifelse__ y (void z) ()));

// Eliminate extra calls to 'void' in generator clauses.
def __do__ (__lambda__ [x] (void y)) = __do__ (__lambda__ [x] y);

// Eliminate extra calls to 'void' in filter clauses.
def __do__ (__lambda__ [x] (__ifelse__ y (void z) ())) =
    __do__ (__lambda__ [x] (__ifelse__ y z ()));

// Any remaining instances reduce to a plain 'do' (this must come last).
def __do__ f = do f;
```

First, note that the `void` function simply throws away its argument and returns `()` instead. The `do` function applies a function to every member of a list (like `map`), but throws away all intermediate results and just returns `()`, which is much more efficient if you don't need those results anyway. These are both defined in the prelude, but we define our own version of `void` here so that we can hook it up to our simplified version of the `__do__` macro.

The `__do__` macro eventually reduces to just a plain `do` call, but applies some optimizations along the way. While the above rules for `__do__` are always valid optimizations for `do`, it's a good idea to use a separate macro here instead of clobbering `do` itself, so that these optimizations do not interfere with calls to `do` in ordinary user code. The prelude handles this in an analogous fashion.

Before we further delve into this example, a few remarks are in order about the way list comprehensions are implemented in Pure. As already mentioned, list comprehensions are just syntactic sugar; the compiler immediately transforms them to an equivalent expression involving only lambdas and a few other list operations. The latter are essentially equivalent to piles of nested filters and maps, but for various reasons they are actually implemented using two special helper operations, `catmap` and `listmap`.

The `catmap` operation combines `map` and `cat`; this is needed, in particular, to accumulate the results of nested generators, such as `[i,j | i = 1..n; j = 1..m]`. The same operation is also used to implement filter clauses, you can see this below in the examples. However, for efficiency simple generators like `[2*i | i = 1..n]` are translated to a `listmap` instead (which is basically just `map`, but works with different aggregate types, so that list comprehensions can draw values from aggregates other than lists, such as matrices).

Now let's see how the rules above transform a list comprehension if we "void" it. (Remember to switch to the `my` namespace when trying the following examples.)

```
> using system;
> using namespace my;
> f = [printf "%g\n" (2^x+1) | x=1..5; x mod 2];
> g = void [printf "%g\n" (2^x+1) | x=1..5; x mod 2];
> show f g
f = catmap (\x -> if x mod 2 then [printf "%g\n" (2^x+1)] else []) (1..5);
g = do (\x -> if x mod 2 then printf "%g\n" (2^x+1) else ()) (1..5);
```

As you can see, the `catmap` got replaced with a `do`, and the list brackets inside the lambda were eliminated as well. These optimizations are just what's needed to make this code go essentially as fast as a `for` loop in traditional programming languages (up to constant factors, of course). Here's how it looks like when we run the `g` function:

```
> g;
3
9
33
()
```

It's also instructive to have a look at how the above macro rules work in concert to rewrite a "voided" comprehension. To these ends, you can rerun the right-hand side of `g` with some tracing enabled, as follows (we omit the tracing output here for brevity):

```
> trace -m my::void
> void [printf "%g\n" (2^x+1) | x=1..5; x mod 2];
```

The above optimization rules also take care of nested list comprehensions, since they recurse into the lambda bodies of generator and filter clauses. For instance:

```
> h = void [puts $ str (x,y) | x=1..2; y=1..3];
> show h
h = do (\x -> do (\y -> puts (str (x,y))) (1..3)) (1..2);
```

Again, you should run this with macro tracing enabled to see how the `__do__` macro recurses into the outer lambda body of the list comprehension. Here's the rule which actually does this:

```
def __do__ (__lambda__ [x] y@(catmap _ _)) =
    __do__ $ (__lambda__ [x] (void y));
```

Note that in order to make this work, `__do__` is implemented as a "*quoteargs*" macro so that it can inspect and recurse into the lambda terms in its argument. Also note the `$` on the right-hand side of this rule; this is also implemented as a macro in the prelude. Here the `$` operator is used to forcibly evaluate the macro argument `__lambda__ [x] (void y)`, so that the embedded call to the `void` macro gets expanded. (Without the `$` the argument to `__do__` would be quoted and thus not be evaluated.) A similar rule is used to recurse into embedded filter clauses, as in the example of the function `g` above.

It should be mentioned that, while our version of the `__do__` macro will properly handle most list comprehensions, there is a rather obscure corner case which it still refuses to optimize: outermost filter clauses. For instance, consider:

```
> let c = 2;
> k = void [printf "%g\n" (2^x+1) | c>0; x=1..3];
> show k
k = my::void (if c>0 then listmap (\x -> printf "%g\n" (2^x+1)) (1..3) else []);
```

It's possible to handle this case as well, but we have to go to some lengths to achieve that. The complication here is that we don't want to mess with calls to `void` in ordinary user code, so `void` itself cannot be a "*quoteargs*" macro. But the quoted form of `void`'s argument is needed to detect the "outermost filter clause" situation. The interested reader may refer to the prelude code to see how the prelude implementation of `__do__` uses some helper macros to make this work. Another detail of the full version of `__do__` is the handling of patterns on the left-hand side of generator clauses, which requires some special magic to filter out unmatched list elements; we also omitted this here for brevity.

1.7.7 Reflection

The meta representation of specials discussed in Built-in Macros and Special Expressions is also useful to obtain information about the running program and even modify it. Pure's runtime provides some built-in operations to implement these reflection capabilities, which are comparable in scope to what the Lisp programming language offers.

Specifically, the get_fundef function allows you to retrieve the definition of a global Pure function. Given the symbol denoting the function, get_fundef returns the list of rewriting rules implementing the functions, using the same `lhs --> rhs` format used by the __case__, __when__ and __with__ macros discussed above. For instance:

```
> fact n = 1 if n<=1;
>        = n*fact (n-1) otherwise;
> get_fundef fact;
[(fact n-->1 __if__ n<=1),(fact n-->n*fact (n-1))]
```

Defining a new function or extending an existing function definition can be done just as easily, using the add_fundef function:

```
> add_fundef $ '[(fib n-->1 __if__ n<=1),(fib n-->fib (n-2)+fib (n-1))];
()
> show fib
fib n = 1 if n<=1;
fib n = fib (n-2)+fib (n-1);
> map fib (0..10);
[1,1,2,3,5,8,13,21,34,55,89]
```

Note that, to be on the safe side, we quoted the rule list passed to add_fundef to prevent premature evaluation of symbols used in the rules. This is necessary because add_fundef is an ordinary function, not a macro. (Of course, you could easily define a macro which would take care of this, if you like. We leave this as an exercise to the reader.)

Also note that add_fundef doesn't override existing function definitions. It simply keeps on adding rules to the current program, just as if you typed the equations at the command prompt of the interpreter. It is possible to delete individual equations with del_fundef:

```
> del_fundef $ '(fib n-->fib (n-2)+fib (n-1));
()
> show fib
fib n = 1 if n<=1;
```

Moreover, the clearsym function allows you to completely get rid of an existing function:

```
> clearsym fib 0;
()
> show fib
> fib 9;
fib 9
```

There's also a companion function, globsym, which enables you to get a list of defined symbols which match a given glob pattern:

```
> globsym "fact" 0;
[fact]
> globsym "*" 0;
[(!),(!!),(#),($),($$),...]
> #globsym "*" 0;
304
```

Note that `globsym` also returns symbols defined as types, macros, variables or constants. But we can easily check for a given type of symbol by using the appropriate function to retrieve the rules defining the symbol, and filter out symbols with an empty rule list:

```
> #[sym | sym = globsym "*" 0; ~null (get_fundef sym)];
253
```

Pure also provides the operations `get_typedef`, `get_macdef`, `get_vardef` and `get_constdef`, which are completely analogous to `get_fundef`, but return the definitions of types, macros, (global) variables and constants. Note that in the latter two cases the rule list takes the form `[var-->val]` if the symbol is defined, `[]` if it isn't.

For instance, let's check the definition of the `$` macro (cf. Optimization Rules) and the `list` type (cf. Recursive Types):

```
> get_macdef ($);
[f$x-->f x]
> get_typedef list;
[(list []-->1),(list (_:_)-->1)]
```

Or let's lists all global variables along with their values:

```
> catmap get_vardef (globsym "*" 0);
[(argc-->0),(argv-->[]),(compiling-->0),
(sysinfo-->"x86_64-unknown-linux-gnu"),(version-->"0.56")]
```

The counterparts of `add_fundef` and `del_fundef` are provided as well. Not very surprisingly, they are named `add_typedef`, `del_typedef`, etc. For instance:

```
> add_vardef ['x-->3*33];
()
> show x
let x = 99;
> del_vardef ('x);
()
> show x
```

The above facilities should cover most metaprogramming needs. For even more exotic requirements, you can also use the `eval` and `evalcmd` primitives to execute arbitrary Pure code in text form; please see the *Pure Library Manual* for details.

Finally, a word of caution: The use of `add_fundef`, `del_fundef` and similar operations to modify a running program breaks referential transparency and hence these functions should be used with care. Moreover, at present the JIT compiler doesn't support truly self-modifying code (i.e., functions modifying themselves while they're executing); this results in undefined behaviour. Also, note that none of the inspection and mutation capabilities provided by these operations will work in batch-compiled programs, please check the Batch Compilation section for details.

1.8 Exception Handling

Pure also offers a useful exception handling facility. To raise an exception, you just invoke the built-in function `throw` with the value to be thrown as the argument. Exceptions are caught with the built-in special form `catch` which is invoked as follows:

`catch` handler x
: Catch an exception. The first argument denotes the exception handler (a function to be applied to the exception value). The second (call-by-name) argument is the expression to be evaluated.

For instance:

```
> catch error (throw hello_world);
error hello_world
```

Exceptions are also generated by the runtime system if the program runs out of stack space, when a guard does not evaluate to a truth value, and when the subject term fails to match the pattern in a pattern-matching lambda abstraction, or a `let`, `case` or `when` construct. These types of exceptions are reported using the symbols `stack_fault`, `failed_cond` and `failed_match`, respectively, which are declared as nonfix symbols in the standard prelude. You can use `catch` to handle these kinds of exceptions just like any other. For instance:

```
> fact n = if n>0 then n*fact(n-1) else 1;
> catch error (fact foo);
error failed_cond
> catch error (fact 100000);
error stack_fault
```

(You'll only get the latter kind of exception if the interpreter does stack checks, see the discussion of the `PURE_STACK` environment variable in Stack Size and Tail Recursion.)

Unhandled exceptions are reported by the interpreter with a corresponding error message:

```
> fact foo;
<stdin>, line 2: unhandled exception 'failed_cond' while evaluating 'fact foo'
```

Note that since the right-hand side of a type definition (cf. Type Rules) is just ordinary Pure code, it may be susceptible to exceptions, too. Such exceptions are reported or caught just like any other. In particular, if you want to make a type definition just fail silently in case of an exception, you'll have to wrap it up in a suitable `catch` clause:

```
> foo x = throw foo; // dummy predicate which always throws an exception
> type bar x = foo x;
> type baz x = catch (cst false) (foo x);
> test_bar x::bar = x;
> test_baz x::baz = x;
> test_bar ();
<stdin>, line 6: unhandled exception 'foo' while evaluating 'test_bar ()'
> test_baz ();
test_baz ()
```

Exceptions also provide a way to handle asynchronous signals. Pure's system module provides symbolic constants for common POSIX signals and also defines the operation `trap` which lets you rebind any signal to a signal exception. For instance, the following lets you handle the `SIGQUIT` signal:

```
> using system;
> trap SIG_TRAP SIGQUIT;
```

You can also use `trap` to just ignore a signal or revert to the system's default handler (which might take different actions depending on the type of signal, see signal(7) for details):

```
> trap SIG_IGN SIGQUIT; // signal is ignored
> trap SIG_DFL SIGQUIT; // reinstalls the default signal handler
```

Note that when the interpreter runs interactively, for convenience most standard termination signals (`SIGINT`, `SIGTERM`, etc.) are already set up to produce corresponding Pure exceptions of the form `signal SIG` where `SIG` is the signal number. If a script is to be run non-interactively then you'll have to do this yourself (otherwise most signals will terminate the program).

While exceptions are normally used to report abnormal error conditions, they also provide a way to implement non-local value returns. For instance, here's a variation of our n queens algorithm (cf. List Comprehensions) which only returns the first solution. Note the use of `throw` in the recursive search routine to bail out with a solution as soon as we found one. The value thrown there is caught in the main routine. Also note the use of `void` in the second equation of `search`. This effectively turns the list comprehension into a simple loop which suppresses the normal list result and just returns `()` instead. Thus, if no value gets thrown then the function regularly returns with `()` to indicate that there is no solution.

```
queens n       = catch reverse (search n 1 []) with
  search n i p = throw p if i>n;
               = void [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = ~any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

E.g., let's compute a solution for a standard 8x8 board:

```
> queens 8;
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

1.9 Standard Library

Pure comes with a collection of Pure library modules, which includes the standard prelude (loaded automatically at startup time) and some other modules which can be loaded explicitly with a `using` clause. The prelude offers the necessary functions to work with the built-in types (including arithmetic and logical operations) and to do most kind of list processing you can find in ML- and Haskell-like languages. It also provides a collection of basic string

and matrix operations. Please refer to the *Pure Library Manual* for details on the provided operations. Here is a very brief summary of some of the prelude operations which, besides the usual arithmetic and logical operators, are probably used most frequently:

x+y
: The arithmetic + operation is also used to denote list and string concatenation in Pure.

x:y
: This is the list-consing operation. `x` becomes the head of the list, `y` its tail. As '`:`' is a constructor symbol, you can use it in patterns on the left hand side of rewriting rules.

x..y
: Constructs arithmetic sequences. `x:y..z` can be used to denote sequences with arbitrary stepsize `y-x`. Infinite sequences can be constructed using an infinite bound (i.e., `inf` or `-inf`). E.g., `1:3..inf` denotes the stream of all odd integers starting at 1.

x,y
: This is the pair constructor, used to create tuples of arbitrary sizes. Tuples provide an alternative way to represent aggregate values in Pure. In contrast to lists, tuples are always "flat", so that `(x,y),z` and `x,(y,z)` denote the same triple `x,y,z`. (This is explained in more detail in the Primary Expressions section.)

#x
: The size (number of elements) of the list, tuple, matrix or string `x`. In addition, `dim x` yields the dimensions (number of rows and columns) of a matrix.

x!y
: This is Pure's indexing operation, which applies to lists, tuples, matrices and strings. Note that all indices in Pure are zero-based, thus `x!0` and `x!(#x-1)` are the first and last element of `x`. In the case of matrices, the subscript may also be a pair of row and column indices, such as `x!(1,2)`.

x!!ys
: This is the "slicing" operation, which returns the list, tuple, matrix or string of all `x!y` while `y` runs through the elements of the list or matrix `ys`. Thus, e.g., `x!!(i..j)` returns all the elements between `i` and `j` (inclusive). Indices which fall outside the valid index range are quietly discarded. The index range `ys` may contain any number of indices (also duplicates), in any order. Thus `x!![0|i=1..n]` returns the first element of `x` `n` times, and, if `ys` is a permutation of the range `0..#x-1`, then `x!!ys` yields the corresponding permutation of the elements of `x`. In the case of matrices the index range may also contain two-dimensional subscripts, or the index range itself may be specified as a pair of row/column index lists such as `x!!(i..j,k..l)`.

The prelude also offers support operations for the implementation of list and matrix comprehensions, as well as the customary list operations like `head`, `tail`, `drop`, `take`, `filter`, `map`, `foldl`, `foldr`, `scanl`, `scanr`, `zip`, `unzip`, etc., which make list programming so much fun in modern FPLs. In Pure, these also work on strings as well as matrices, although, for reasons of efficiency, these data structures are internally represented as arrays.

Besides the prelude, Pure's standard library also comprises a growing number of additional library modules which we can only mention in passing here. In particular, the `math` module

provides additional mathematical functions as well as Pure's complex and rational number data types. Common container data structures like sets and dictionaries are implemented in the `set` and `dict` modules, among others. Moreover, the system interface can be found in the `system` module. In particular, this module also provides operations to do basic C-style I/O, including `printf` and `scanf`.

1.10 C Interface

Pure makes it very easy to call C functions (as well as functions in a number of other languages supported by the GNU compiler collection). To call an existing C function, you just need an `extern` declaration of the function, as described below. By these means, all functions in the standard C library and the Pure runtime are readily available to Pure scripts. Functions can also be loaded from dynamic libraries and LLVM bitcode files at runtime. In the latter case, you don't even need to write any `extern` declarations, the interpreter will do that for you. As of Pure 0.45, you can also add inline C/C++ and Fortran code to your Pure scripts and have the Pure interpreter compile them on the fly, provided that you have the corresponding compilers from the LLVM project installed.

In some cases you will still have to rely on big and complicated third-party and system libraries which aren't readily available in bitcode form. It goes without saying that writing all the `extern` declarations for such libraries can be a daunting task. Fortunately, there is a utility to help with this, by extracting the `extern` declarations automatically from C headers. Please see External C Functions in the Caveats and Notes section for details.

1.10.1 Extern Declarations

To access an existing C function in Pure, you need an `extern` declaration of the function, which is a simplified kind of C prototype. The syntax of these declarations is described by the following grammar rules:

```
extern_decl  ::=  [scope] "extern" prototype ("," prototype) ";"
prototype    ::=  c_type identifier "(" [parameters | "..."]  ")"  ["=" identifier]
parameters   ::=  parameter ("," parameter)* ["," "..."]
parameter    ::=  c_type [identifier]
c_type       ::=  identifier "*"*
```

Extern functions can be called in Pure just like any other. For instance, the following commands, entered interactively in the interpreter, let you use the `sin` function from the C library (of course you could just as well put the `extern` declaration into a script):

```
> extern double sin(double);
> sin 0.3;
0.29552020666134
```

An `extern` declaration can also be prefixed with a `public`/`private` scope specifier:

```
private extern double sin(double);
```

Multiple prototypes can be given in one `extern` declaration, separating them with commas:

```
extern double sin(double), double cos(double), double tan(double);
```

For clarity, the parameter types can also be annotated with parameter names (these only serve informational purposes and are for the human reader; they are effectively treated as comments by the compiler):

```
extern double sin(double x);
```

Pointer types are indicated by following the name of the element type with one or more asterisks, as in C. For instance:

```
> extern char* strchr(char *s, int c);
> strchr "foo bar" (ord "b");
"bar"
```

As you can see in the previous example, some pointer types get special treatment, allowing you to pass certain kinds of Pure data (such as Pure strings as `char*` in this example). This is discussed in more detail in C Types below.

The interpreter makes sure that the parameters in a call match; if not, then by default the call is treated as a normal form expression:

```
> extern double sin(double);
> sin 0.3;
0.29552020666134
> sin 0;
sin 0
```

This gives you the opportunity to augment the external function with your own Pure equations. To make this work, you have to make sure that the `extern` declaration of the function comes first. For instance, we might want to extend the `sin` function with a rule to handle integers:

```
> sin x::int = sin (double x);
> sin 0;
0.0
```

Sometimes it is preferable to replace a C function with a wrapper function written in Pure. In such a case you can specify an **alias** under which the original C function is known to the Pure program, so that you can still call the C function from the wrapper. An alias is introduced by terminating the extern declaration with a clause of the form `= alias`. For instance:

```
> extern double sin(double) = c_sin;
> sin x::double = c_sin x;
> sin x::int = c_sin (double x);
> sin 0.3; sin 0;
0.29552020666134
0.0
```

Aliases are just one way to declare a **synonym** of an external function. As an alternative, you can also declare the C function in a special namespace (cf. Namespaces in the Declarations section):

```
> namespace c;
> extern double sin(double);
> c::sin 0.3;
0.29552020666134
```

Note that the namespace qualification only affects the Pure side; the underlying C function is still called under the unqualified name as usual. The way in which such qualified externs are accessed is the same as for ordinary qualified symbols. In particular, the `using namespace` declaration applies as usual, and you can declare such symbols as `private` if needed. It is also possible to combine a namespace qualifier with an alias:

```
> namespace c;
> extern double sin(double) = mysin;
> c::mysin 0.3;
0.29552020666134
```

In either case, different synonyms of the same external function can be declared in slightly different ways, which makes it possible to adjust the interpretation of pointer values on the Pure side. This is particularly useful for string arguments which, as described below, may be passed both as `char*` (which implies copying and conversion to or from the system encoding) and as `void*` (which simply passes through the character pointers). For instance:

```
> extern char *strchr(char *s, int c) = foo;
> extern void *strchr(void *s, int c) = bar;
> foo "foo bar" 98; bar "foo bar" 98;
"bar"
#<pointer 0x12c2f24>
```

Also note that, as far as Pure is concerned, different synonyms of an external function are really different functions. In particular, they can each have their own set of augmenting Pure equations. For instance:

```
> extern double sin(double);
> extern double sin(double) = mysin;
> sin === sin;
1
> sin === mysin;
0
> sin 1.0; mysin 1.0;
0.841470984807897
0.841470984807897
> sin x::int = sin (double x);
> sin 1; mysin 1;
0.841470984807897
mysin 1
```

1.10.2 Variadic C Functions

Variadic C functions are declared as usual by terminating the parameter list with an ellipsis (`...`):

```
> extern int printf(char*, ...);
> printf "Hello, world\n";
Hello, world
13
```

Note that the variadic prototype is mandatory here, since the compiler needs to know about the optional arguments so that it can generate the proper code to call the function. However, in Pure a function always has a fixed arity, so, as far as Pure is concerned, the function is still treated as if it had no extra arguments. Thus the above declaration only allows you to call `printf` with a single argument.

To make it possible to pass optional arguments to a variadic function, you must explicitly give the (non-variadic) prototypes with which the function is to be called. To these ends, the additional prototypes are declared as synonyms of the original variadic function. This works because the compiler only checks the non-variadic parameters for conformance. For instance:

```
> extern int printf(char*, char*) = printf_s;
> printf_s "Hello, %s\n" "world";
Hello, world
13
> extern int printf(char*, int) = printf_d;
> printf_d "Hello, %d\n" 99;
Hello, 99
10
```

1.10.3 C Types

As indicated in the previous section, the data types in `extern` declarations are either C type names or pointer types derived from these. The special `expr*` pointer type is simply passed through; this provides a means to deal with Pure data in C functions in a direct fashion. For all other C types, Pure values are "marshalled" (converted) from Pure to C when passed as arguments to C functions, and the result returned by the C function is then converted back from C to Pure. All of this is handled by the runtime system in a transparent way, of course.

Note that, to keep things simple, Pure does *not* provide any notations for C structs or function types, although it is possible to represent pointers to such objects using `void*` or some other appropriate pointer types. In practice, this simplified system should cover most kinds of calls that need to be done when interfacing to C libraries, but there are ways to work around these limitations if you need to access C structs or call back from C to Pure, see External C Functions in the Caveats and Notes section for details.

Basic C Types

Pure supports the usual range of basic C types: `void`, `bool`, `char`, `short`, `int`, `long`, `float`, `double`, and converts between these and the corresponding Pure data types (machine ints, bigints and double values) in a straightforward way.

The `void` type is only allowed in function results. It is converted to the empty tuple `()`.

Both `float` and `double` are supported as floating point types. Single precision `float` arguments and return values are converted from/to Pure's double precision floating point numbers.

A variety of C integer types (`bool`, `char`, `short`, `int`, `long`) are provided which are converted from/to the available Pure integer types in a straightforward way. In addition, the synonyms `int8`, `int16` and `int32` are provided for `char`, `short` and `int`, respectively, and `int64` denotes 64 bit integers (a.k.a. ISO C99 `long long`). Note that `long` is equivalent to `int32` on 32 bit systems, whereas it is the same as `int64` on most 64 bit systems. To make it easier to interface to various system routines, there's also a special `size_t` integer type which usually is 4 bytes on 32 bit and 8 bytes on 64 bit systems.

All integer parameters take both Pure ints and bigints as actual arguments; truncation or sign extension is performed as needed, so that the C interface behaves as if the argument was "cast" to the C target type. Returned integers use the smallest Pure type capable of holding the result, i.e., int for the C `char`, `short` and `int` types, bigint for `int64`.

Pure considers all integers as signed quantities, but it is possible to pass unsigned integers as well (if necessary, you can use a bigint to pass positive values which are too big to fit into a machine int). Also note that when an unsigned integer is returned by a C routine, which is too big to fit into the corresponding signed integer type, it will "wrap around" and become negative. In this case, depending on the target type, you can use the `ubyte`, `ushort`, `uint`, `ulong` and `uint64` functions provided by the prelude to convert the result back to an unsigned quantity.

Pointer Types

The use of pointer types is also fairly straightforward, but Pure has some special rules for the conversion of certain pointer types which make it easy to pass aggregate Pure data to and from C routines, while also following the most common idioms for pointer usage in C. The following types of pointers are recognized both as arguments and return values of C functions.

Bidirectional pointer conversions:

- `char*` is used for string arguments and return values which are converted from Pure's internal utf-8 based string representation to the system encoding and vice versa. (Thus a C routine can never modify the raw Pure string data in-place; if this is required then you'll have to pass the string argument as a `void*`, see below.)
- `void*` is for any generic pointer value, which is simply passed through unchanged. When used as an argument, you can also pass Pure strings, matrices and bigints. In

this case the raw underlying data pointer (`char*` in the case of strings, `int*`, `double*` or `expr*` in the case of numeric and symbolic matrices, and the GMP type `mpz_t` in the case of bigints) is passed, which allows the data to be modified in place (with care). In particular, passing bigints as `void*` makes it possible to call most GMP integer routines directly from Pure.

- `dmatrix*`, `cmatrix*` and `imatrix*` allow you to pass numeric Pure matrices of the appropriate types (double, complex, int). Here a pointer to the underlying GSL matrix structure is passed (not just the data itself). This makes it possible to transfer GSL matrices between Pure and GSL routines in a direct fashion without any overhead. (For convenience, there are also some other pointer conversions for marshalling matrix arguments to numeric C vectors, which are described in Pointers and Matrices below.)
- `expr*` is for any kind of Pure value. A pointer to the expression node is passed to or from the C function. This type is to be used for C routines which are prepared to deal with pristine Pure data, using the corresponding functions provided by the runtime. You can find many examples of this in the standard library.

All other pointer types are simply taken at face value, allowing you to pass Pure pointer values as is, without any conversions. This also includes pointers to arbitrary named types which don't have a predefined meaning in Pure, such as `FILE*`. As of Pure 0.45, the interpreter keeps track of the actual names of all pointer types and checks (at runtime) that the types match in an external call, so that you can't accidentally get a core dump by passing, say, a `FILE*` for a `char*`. (The call will then simply fail and yield a normal form, which gives you the opportunity to hook into the function with your own Pure definitions which may supply any desired data conversions.) Typing information about pointer values is also available to Pure scripts by means of corresponding library functions, please see the *Tagged Pointers* section in the *Pure Library Manual* for details.

Pointers and Matrices

The following additional pointer conversions are provided to deal with Pure matrix values in arguments of C functions, i.e., on the input side. These enable you to pass Pure matrices for certain kinds of C vectors. Note that in any case, you can also simply pass a suitable plain pointer value instead. Also, these types aren't special in return values, where they will simply yield a pointer value (with the exception of `char*` which gets special treatment as explained in the previous subsection). Thus you will have to decode such results manually if needed. The standard library provides various routines to do this, please see the *String Functions* and *Matrix Functions* sections in the *Pure Library Manual* for details.

Numeric pointer conversions (input only):

- `char*`, `short*`, `int*`, `int64*`, `float*`, `double*` can be used to pass numeric matrices as C vectors. This kind of conversion passes just the matrix data (not the GSL matrix structure, as the `dmatrix*` et al conversions do) and does conversions between integer or floating point data of different sizes on the fly. You can either pass an int matrix as a `char*`, `short* int*` or `int64*` argument, or a double or complex matrix as a `float*`

or `double*` argument (complex values are then represented as two separate double numbers, first the real, then the imaginary part, for each matrix element).

- `char**`, `short**`, `int**`, `int64**`, `float**`, `double**` provide yet another way to pass numeric matrix arguments. This works analogously to the numeric vector conversions above, but here a temporary C vector of pointers is passed to the C function, whose elements point to the rows of the matrix.

Argv-style conversions (input only):

- `char**` and `void**` can be used to pass `argv`-style vectors as arguments to C functions. In this case, the Pure argument must be a symbolic vector of strings or generic pointer values. `char**` converts the string elements to the system encoding, whereas `void**` passes through character string data and other pointers unchanged (and allows in-place modification of the data). A temporary C vector of these elements is passed to the C function, which is always `NULL`-terminated and can thus be used for almost any purpose which requires such `argv`-style vectors.

Note that in the numeric pointer conversions, the matrix data is passed "per reference" to C routines, i.e., the C function may modify the data "in place". This is true even for target data types such as `short*` or `float**` which involve automatic conversions and hence need temporary storage. In this case the data from the temporary storage is written back to the original matrix when the function returns, to maintain the illusion of in-place modification. Temporary storage is also needed when the GSL matrix has the data in non-contiguous storage. You may want to avoid this if performance is critical, by always using "packed" matrices (see `pack` in *Matrix Functions*) of the appropriate types.

Pointer Examples

Let's finally have a look at some instructive examples to explain some of the trickier pointer types.

First, the matrix pointer types `dmatrix*`, `cmatrix*` and `imatrix*` can be used to pass double, complex double and int matrices to GSL functions taking pointers to the corresponding GSL types (`gsl_matrix`, `gsl_matrix_complex` and `gsl_matrix_int`) as arguments or returning them as results. (Note that there is no special marshalling of Pure's symbolic matrix type, as these aren't supported by GSL anyway.) Also note that matrices are always passed by reference. Thus, if you need to pass a matrix as an output parameter of a GSL matrix routine, you should either create a zero matrix or a copy of an existing matrix to hold the result. The prelude provides various operations for that purpose (in particular, see the `dmatrix`, `cmatrix`, `imatrix` and `pack` functions in matrices.pure). For instance, here is how you can quickly wrap up GSL's double matrix addition function in a way that preserves value semantics:

```
> using "lib:gsl";
> extern int gsl_matrix_add(dmatrix*, dmatrix*);
> x::matrix + y::matrix = gsl_matrix_add x y $$ x when x = pack x end;
> let x = dmatrix {1,2,3}; let y = dmatrix {2,3,2}; x; y; x+y;
{1.0,2.0,3.0}
```

```
{2.0,3.0,2.0}
{3.0,5.0,5.0}
```

Most GSL matrix routines can be wrapped in this fashion quite easily. A ready-made GSL interface providing access to all of GSL's numeric functions is in the works; please check the Pure website for details.

For convenience, it is also possible to pass any kind of numeric matrix for a `char*`, `short*`, `int*`, `int64*`, `float*` or `double*` parameter. This requires that the pointer and the matrix type match up; conversions between `char`, `short`, `int64` and `int` data and, likewise, between `float` and `double` are handled automatically, however. For instance, here is how you can call the `puts` routine from the C library with an int matrix encoding the string `"Hello, world!"` as byte values (ASCII codes):

```
> extern int puts(char*);
> puts {72,101,108,108,111,44,32,119,111,114,108,100,33,0};
Hello, world!
14
```

Pure 0.45 and later also support `char**`, `short**`, `int**`, `int64**`, `float**` and `double**` parameters which encode a matrix as a vector of row pointers instead. This kind of matrix representation is often found in audio and video processing software (where the rows of the matrix might denote different audio channels, display lines or video frames), but it's also fairly convenient to do any kind of matrix processing in C. For instance, here's how to do matrix multiplication (the naive algorithm):

```
void matmult(int n, int l, int m, double **x, double **y, double **z)
{
  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++) {
      z[i][j] = 0.0;
      for (k = 0; k < l; k++)
        z[i][j] += x[i][k]*y[k][j];
    }
}
```

As you can see, this multiplies a `n` times `l` matrix `x` with a `l` times `m` matrix `y` and puts the result into the `n` times `m` matrix `z`:

```
> extern void matmult(int, int, int, double**, double**, double**);
> let x = {0.11,0.12,0.13;0.21,0.22,0.23};
> let y = {1011.0,1012.0;1021.0,1022.0;1031.0,1032.0};
> let z = dmatrix (2,2);
> matmult 2 3 2 x y z $$ z;
{367.76,368.12;674.06,674.72}
```

Also new in Pure 0.45 is the support for passing `argv`-style vectors as arguments. For instance, here is how you can use `fork` and `execvp` to implement a poor man's version of the C `system` function. (This is Unix-specific and doesn't do much error-checking, but you get the idea.)

```
extern int fork();
extern int execvp(char *path, char **argv);
extern int waitpid(int pid, int *status, int options);

system cmd::string = case fork of
  // child: execute the program, bail out if error
  0 = execvp "/bin/sh" {"/bin/sh","-c",cmd} $$ exit 1;
  // parent: wait for the child and return its exit code
  pid = waitpid pid status 0 $$ status!0 >> 8
        when status = {0} end if pid>=0;
end;

system "echo Hello, world!";
system "ls -l *.pure";
system "exit 1";
```

1.10.4 Importing Dynamic Libraries

By default, external C functions are resolved by the LLVM runtime, which first looks for the symbol in the C library and Pure's runtime library (or the interpreter executable, if the interpreter was linked statically). Thus all C library and Pure runtime functions are readily available in Pure programs. Other functions can be provided by adding them to the runtime, or by linking them into the runtime or the interpreter executable. Better yet, you can just "dlopen" shared libraries at runtime with a special form of the `using` clause:

```
using "lib:libname[.ext]";
```

For instance, if you want to call the functions from library libxyz directly from Pure:

```
using "lib:libxyz";
```

After this declaration the functions from the given library will be ready to be imported into your Pure program by means of corresponding `extern` declarations.

Shared libraries opened with using clauses are searched for in the same way as source scripts (see section Modules and Imports above), using the `-L` option and the `PURE_LIBRARY` environment variable in place of `-I` and `PURE_INCLUDE`. If the library isn't found by these means, the interpreter will also consider other platform-specific locations searched by the dynamic linker, such as the system library directories and `LD_LIBRARY_PATH` on Linux. The necessary filename suffix (e.g., .so on Linux or .dll on Windows) will be supplied automatically when needed. Of course you can also specify a full pathname for the library if you prefer that. If a library file cannot be found, or if an `extern` declaration names a function symbol which cannot be resolved, an appropriate error message is printed.

1.10.5 Importing LLVM Bitcode

As of Pure 0.44, the interpreter also provides a direct way to import LLVM bitcode modules in Pure scripts. The main advantage of this method over the "plain" C interface explained

above is that the bitcode loader knows all the call interfaces and generates the necessary `extern` declarations automatically. This is more than just a convenience, as it also eliminates at least some of the mistakes in `extern` declarations that may arise when importing functions manually from dynamic libraries.

Note: The facilities described below require that you have an LLVM-capable C/C++ compiler installed. The available options right now are clang, llvm-gcc and dragonegg. Please check the Pure *installation instructions* on how to get one of these (or all of them) up and running. Note that clang and llvm-gcc are standalone compilers, while dragonegg is supplied as a gcc plugin which hooks into your existing system compiler (gcc 4.5 or later is required for that). Any of these enable you to compile C/C++ source to LLVM assembler or bitcode. The clang compiler is recommended for C/C++ development, as it offers faster compilation times and has much better diagnostics than gcc. On the other hand, llvm-gcc and dragonegg have the advantage that they also support alternative frontends so that you can compile Fortran and Ada code as well. (But note that, as of LLVM 3.x, llvm-gcc is not supported any more.)

LLVM bitcode is loaded in a Pure script using the following special format of the `using` clause:

```
using "bc:modname[.bc]";
```

(Here the `bc` tag indicates a bitcode file, and the default `.bc` bitcode filename extension is supplied automatically. Also, the bitcode file is searched for on the usual library search path.)

That's it, no explicit `extern` declarations are required on the Pure side. The Pure interpreter automatically creates `extern` declarations (in the current namespace) for all the external functions defined in the LLVM bitcode module, and generates the corresponding wrappers to make the functions callable from Pure. (This also works when batch-compiling a Pure script. In this case, the bitcode file actually gets linked into the output code, so the loaded bitcode module only needs to be present at compile time.)

By default the imported symbols will be public. You can also specify the desired scope of the symbols explicitly, by placing the `public` or `private` keyword before the module name. For instance:

```
using private "bc:modname";
```

You can also import the same bitcode module several times, possibly in different namespaces. This will not actually reload the module, but it will create synonyms for the external functions in different namespaces:

```
namespace foo;
using "bc:modname";
namespace bar;
using private "bc:modname";
```

You can load any number of bitcode modules along with shared libraries in a Pure script, in

any order. The JIT will try to satisfy external references in modules and libraries from other loaded libraries and bitcode modules. This is deferred until the code is actually JIT-compiled, so that you can make sure beforehand that all required libraries and bitcode modules have been loaded. If the JIT fails to resolve a function, the interpreter will print its name and also raise an exception at runtime when the function is being called from other C code. (You can then run your script in the debugger to locate the external visible in Pure from which the unresolved function is called.)

Let's take a look at a concrete example to see how this actually works. Consider the following C code which defines a little function to compute the greatest common divisor of two (machine) integers:

```
int mygcd(int x, int y)
{
  if (y == 0)
    return x;
  else
    return mygcd(y, x%y);
}
```

Let's say that this code is in the file `mygcd.c`, then you'd compile it to a bitcode module using clang as follows:

```
clang -emit-llvm -c mygcd.c -o mygcd.bc
```

Note that the `-emit-llvm -c` options instruct clang to build an LLVM bitcode module. Of course, you can also add optimizations and other options to the compile command as desired.

Using dragonegg is somewhat more involved, as it doesn't provide a direct way to produce a bitcode file yet. However, you can create an LLVM assembler file which can then be translated to bitcode using the llvm-as program as follows:

```
gcc -fplugin=dragonegg -flto -S mygcd.c -o mygcd.ll
llvm-as mygcd.ll -o mygcd.bc
```

(Note that the `-fplugin` option instructs gcc to use the dragonegg plugin, which in conjunction with the `-flto` flag switches it to LLVM output. Please check the dragonegg website for details.)

In either case, you can now load the resulting bitcode module and run the `mygcd` function in the Pure interpreter simply as follows:

```
> using "bc:mygcd";
> mygcd 75 105;
15
```

To actually see the generated `extern` declaration of the imported function, you can use the interactive `show` command:

```
> show mygcd
extern int mygcd(int, int);
```

Some more examples showing how to use the bitcode interface can be found in the Pure sources. In particular, the interface also works with Fortran (using llvm-gfortran or gfortran with dragonegg), and there is special support for interfacing to Grame's functional DSP programming language Faust (the latter uses a special variant of the bitcode loader, which is selected with the `dsp` tag in the `using` clause). Further details about these can be found below.

Please note that at this time the LLVM bitcode interface is still somewhat experimental, and there are some known limitations:

- LLVM doesn't distinguish between `char*` and `void*` in bitcode, so all `void*` parameters and return values in C code will be promoted to `char*` on the Pure side. Also, pointers to types which neither have a predefined meaning in Pure nor a proper type name in the bitcode file, will become a generic pointer type (`void*`, `void**`, etc.) in Pure. If this is a problem then you can just redeclare the corresponding functions under a synonym *after* loading the bitcode module, giving the proper argument and result types (see Extern Declarations above). For instance:

```
> using "bc:foo";
> show foo
extern char* foo(char*);
> extern void *foo(void*) = myfoo;
> show myfoo
extern void* foo(void*) = myfoo;
```

- The bitcode interface is limited to the same range of C types as Pure's plain C interface. In practice, this should cover most C code, but it's certainly possible that you run into unsupported types for arguments and return values. The compiler will then print a warning; the affected functions will still be linked in, but they will not be callable from Pure. Also note that calling conventions for passing C structs *by value* depend on the host ABI, so you should have a look at the resulting `extern` declaration (using `show`) to determine how the function is actually to be called from Pure.

1.10.6 Inline Code

Instead of manually compiling source files to bitcode modules, you can also just place the source code into a Pure script, enclosing it in `%< ... %>`. (Optionally, the opening brace may also be preceded with a `public` or `private` scope specifier, which is used in the same way as the scope specifier following the `using` keyword when importing bitcode files.)

For instance, here is a little script showing inline code for the `mygcd` function from the previous subsection:

```
%<
int mygcd(int x, int y)
{
  if (y == 0)
    return x;
  else
```

```
    return mygcd(y, x%y);
}
%>

mygcd 75 105;
```

The interpreter automatically compiles the inlined code to LLVM bitcode which is then loaded as usual. (Of course, this will only work if you have the corresponding LLVM compilers installed.) This method has the advantage that you don't have to write a Makefile and you can create self-contained Pure scripts which include all required external functions. The downside is that the inline code sections will have to be recompiled every time you run the script with the interpreter which may considerably increase startup times. If this is a problem then it's usually better to import a separate bitcode module instead (see Importing LLVM Bitcode), or batch-compile your script to an executable (see Batch Compilation).

At present, C, C++, Fortran and Faust are supported as foreign source languages, with clang, clang++, gfortran (with the dragonegg plugin) and faust as the corresponding (default) compilers. C is the default language. The desired source language can be selected by placing an appropriate tag into the inline code section, immediately after the opening brace. (The tag is removed before the code is submitted to compilation.) For instance:

```
%< -*- Fortran90 -*-
function fact(n) result(p)
  integer n, p
  p = 1
  do i = 1, n
     p = p*i
  end do
end function fact
%>

fact n::int = fact_ {n};
map fact (1..10);
```

As indicated, the language tag takes the form `-*- lang -*-` where `lang` can currently be any of `c`, `c++`, `fortran` and `dsp` (the latter indicates the Faust language). Case is insignificant here, so you can also write `C`, `C++`, `Fortran`, `DSP` etc. For the `fortran` tag, you may also have to specify the appropriate language standard, such as `fortran90` which is used in the example above. The language tag can also be followed by a module name, using the format `-*- lang:name -*-`. This is optional for all languages except Faust (where the module name specifies the namespace for the interface routines of the Faust module; see Interfacing to Faust below). So, e.g., a Faust DSP named `test` would be specified with a `dsp:test` tag. Case is *significant* in the module name.

The Pure interpreter has some built-in knowledge on how to invoke the LLVM compilers to produce a working bitcode file ready to be loaded by the interpreter, so the examples above should work out of the box if you have the required compilers installed on your `PATH`. However, there are also some environment variables you can set for customization purposes. Specifically, `PURE_CC` is the command to invoke the C compiler. This variable lets you specify the exact name of the executable along with any debugging and optimization options that

you may want to add. Likewise, `PURE_CXX`, `PURE_FC` and `PURE_FAUST` are used for the C++, Fortran and Faust compilers, respectively.

For instance, if you prefer to use llvm-gcc as your C compiler, and you'd like to invoke it with the `-O3` optimization option, you would set `PURE_CC` to `"llvm-gcc -O3"`. (To verify the settings you made, you can have the interpreter echo the compilation commands which are actually executed, by running Pure with the `-v0100` option, see Verbosity and Debugging Options. Also note that the options necessary to produce LLVM bitcode will be added automatically, so you don't have to specify these.)

Beginning with Pure 0.48, the dragonegg gcc plugin is also fully supported. To make this work, you need to explicitly specify the name of the plugin in the compilation command, so that the Pure interpreter can add the proper set of options needed for bitcode compilation. For instance:

```
PURE_CC="gcc -fplugin=dragonegg -O3"
```

Some further details on the bitcode support for specific target languages can be found in the subsections below.

1.10.7 Interfacing to C++

Interfacing to C++ code requires additional preparations because of the name mangling performed by C++ compilers. Usually, you won't be able to call C++ functions and methods directly, so you'll have to expose the required functionality using functions with C binding (`extern "C"`). For instance, the following example shows how to work with STL maps from Pure.

```
%< -*- C++ -*-

#include <pure/runtime.h>
#include <string>
#include <map>

// An STL map mapping strings to Pure expressions.

using namespace std;
typedef map<string,pure_expr*> exprmap;

// Since we can't directly deal with C++ classes in Pure, provide some C
// functions to create, destroy and manipulate these objects.

extern "C" exprmap *map_create()
{
  return new exprmap;
}

extern "C" void map_add(exprmap *m, const char *key, pure_expr *x)
{
  exprmap::iterator it = m->find(string(key));
```

```
  if (it != m->end()) pure_free(it->second);
  (*m)[key] = pure_new(x);
}

extern "C" void map_del(exprmap *m, const char *key)
{
  exprmap::iterator it = m->find(key);
  if (it != m->end()) {
    pure_free(it->second);
    m->erase(it);
  }
}

extern "C" pure_expr *map_get(exprmap *m, const char *key)
{
  exprmap::iterator it = m->find(key);
  return (it != m->end())?it->second:0;
}

extern "C" pure_expr *map_keys(exprmap *m)
{
  size_t i = 0, n = m->size();
  pure_expr **xs = new pure_expr*[n];
  for (exprmap::iterator it = m->begin(); it != m->end(); ++it)
    xs[i++] = pure_string_dup(it->first.c_str());
  pure_expr *x = pure_listv(n, xs);
  delete[] xs;
  return x;
}

extern "C" void map_destroy(exprmap *m)
{
  for (exprmap::iterator it = m->begin(); it != m->end(); ++it)
    pure_free(it->second);
  delete m;
}

%>

// Create the STL map and add a sentry so that it garbage-collects itself.
let m = sentry map_destroy map_create;

// Populate the map with some arbitrary Pure data.
do (\(x=>y) -> map_add m x y) ["foo"=>99, "bar"=>bar 4711L, "baz"=>1..5];

// Query the map.
map_keys m; // => ["bar","baz","foo"]
map (map_get m) (map_keys m); // => [bar 4711L,[1,2,3,4,5],99]

// Delete an element.
map_del m "foo";
map_keys m; // => ["bar","baz"]
```

```
map (map_get m) (map_keys m); // => [bar 4711L,[1,2,3,4,5]]
```

1.10.8 Interfacing to Faust

Faust is a functional dsp (digital signal processing) programming language developed at Grame, which is tailored to the task of generating and transforming streams of numeric data at the sample level. It is typically used to program sound synthesis and audio effect units, but can in fact be employed to process any kind of numeric vector and matrix data. The Faust compiler is capable of generating very efficient code for such tasks which is comparable in performance with carefully handcrafted C routines. Pure's Faust interface lets you use these capabilities in order to process sample data stored in Pure matrices.

Pure's LLVM bitcode loader has some special knowledge about Faust built into it, which makes interfacing to Faust programs simple and efficient. At present, you'll need a special LLVM-capable version of Faust to make this work, which is available under the "faust2" branch in Faust's git repository. Some information on how to get this up and running can be found on the LLVM backend for Faust website.

Note: There's also an alternative interface to Faust which is available as a separate package and works with either Faust2 or the stable Faust version. Please check the *pure-faust* package for details. This package also provides the `faust2` compatibility module which implements the pure-faust API on top of Pure's built-in Faust interface, so that you can also use the operations of this module instead. (The pure-faust API can in fact be more convenient to use in some cases, especially if you want to load a lot of different Faust modules dynamically at runtime.)

The `-lang llvm` option instructs the Faust compiler to output LLVM bitcode. Also, you want to add the `-double` option to make the compiled Faust module use double precision floating point values for samples and control values. So you'd compile an existing Faust module in the source file example.dsp as follows:

```
faust -double -lang llvm example.dsp -o example.bc
```

The `-double` option isn't strictly necessary, but it makes interfacing between Pure and Faust easier and more efficient, since Pure uses `double` as its native floating point format.

Alternatively, you can also use the Faust `pure.c` architecture (included in recent Faust2 revisions and also in the *pure-faust* package) to compile a Faust program to corresponding C source which can then be fed into an LLVM-capable C compiler to produce bitcode which is compatible with Pure's Faust bitcode loader. This is useful, in particular, if you want to make use of special optimization options provided by the C compiler, or if the Faust module needs to be linked against additional C/C++ code. For instance:

```
faust -double -a pure.c -lang c example.dsp -o example.c
clang -emit-llvm -O3 -c example.c -o example.bc
```

A third possibility is to just inline Faust code in a Pure script, as described in the Inline Code section. The compilation step is then handled by the Pure compiler and the `-double` option is added automatically. The `PURE_FAUST` environment variable can be used to specify a custom Faust command to be invoked by the Pure interpreter. This is useful if you'd like to invoke the Faust compiler with some special options, e.g.:

```
PURE_FAUST="faust -single -vec"
```

(Note that you do not have to include the `-lang llvm` option; the inline compiler will supply it automatically.)

Moreover, you can also set the `FAUST_OPT` environment variable to specify any needed post-processing of the output of the Faust compiler; this is typically used to invoke the LLVM `opt` utility in a pipeline, in order to have some additional optimizations performed on the Faust-generated code:

```
FAUST_OPT="| opt -O3"
```

After loading or inlining the Faust module, the Pure compiler makes the interface routines of the Faust module available in its own namespace. Thus, e.g., the interface routines for the example.dsp module will end up in the `example` namespace.

Pure's Faust interface offers another useful feature not provided by the general bitcode interface, namely the ability to reload Faust modules on the fly. If you repeat the import clause for a Faust module, the compiler checks whether the module was modified and, if so, replaces the old module with the new one. Retyping an inline Faust code section has the same effect. This is mainly intended as a convenience for interactive usage, so that you can test different versions of a Faust module without having to restart the Pure interpreter. But it is also put to good use in addon packages like *pd-faust* which allows Faust dsps to be reloaded at runtime.

For instance, consider the following little Faust program, which takes a stereo audio signal as input, mixes the two channels and multiplies the resulting mono signal with a gain value given by a corresponding Faust control variable:

```
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
```

The interface routines of this Faust module look as follows on the Pure side:

```
> show -g example::*
extern void buildUserInterface(struct_dsp_example*, struct_UIGlue*) = example::buildUserInterface;
extern void classInit(int) = example::classInit;
extern void compute(struct_dsp_example*, int, double**, double**) = example::compute;
extern void delete(struct_dsp_example*) = example::delete;
extern void destroy(struct_dsp_example*) = example::destroy;
extern int getNumInputs(struct_dsp_example*) = example::getNumInputs;
extern int getNumOutputs(struct_dsp_example*) = example::getNumOutputs;
extern int getSampleRate(struct_dsp_example*) = example::getSampleRate;
extern expr* info(struct_dsp_example*) = example::info;
extern void init(struct_dsp_example*, int) = example::init;
extern void instanceInit(struct_dsp_example*, int) = example::instanceInit;
```

```
extern expr* meta() = example::meta;
extern void metadata(struct_MetaGlue*) = example::metadata;
extern struct_dsp_example* new() = example::new;
extern struct_dsp_example* newinit(int) = example::newinit;
```

The most important interface routines are `new`, `init` and `delete` (used to create, initialize and destroy an instance of the dsp) and `compute` (used to apply the dsp to a given block of samples). Some useful convenience functions are added by the Pure compiler:

- `newinit` combines `new` and `init`;
- `info` yields pertinent information about the dsp as a Pure tuple containing the number of input and output channels and the Faust control descriptions;
- `meta` yields metadata about the dsp, as declared in the Faust source.

The latter two are provided in a symbolic format ready to be used in Pure; more about that below.

Note that there's usually no need to explicitly invoke the `delete` routine in Pure programs; the Pure compiler makes sure that this routine is added automatically as a finalizer (see `sentry`) to all dsp pointers created through the `new` and `newinit` routines so that dsp instances are destroyed automatically when the corresponding Pure objects are garbage-collected. (If you prefer to do the finalization manually then you must also remove the sentry from the dsp object, so that it doesn't get deleted twice.)

Another point worth mentioning here is that the Pure compiler always generates code that ensures that the Faust dsp instances (the `struct_dsp` pointers) are fully typechecked at runtime. Thus it is only possible to pass a dsp struct pointer to the interface routines of the Faust module it was created with.

Let's have a brief look at how we can actually run a Faust module in Pure to process some audio samples.

Step 1: Load the Faust dsp. This assumes that the Faust source has already been compiled to a bitcode file, as shown above. You can then load the module in Pure as follows:

```
> using "dsp:example";
```

Note that the .bc bitcode extension is supplied automatically. Also note the special `dsp` tag; this tells the compiler that this is a Faust-generated module, so that it does some Faust-specific processing while linking the module.

Alternatively, you can also just inline the code of the Faust module. For the example above, the inline code section looks as follows:

```
%< -*- dsp:example -*-
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
%>
```

You can either add this code to a Pure script, or just type it directly in the Pure interpreter.

Finally, you may want to verify that the module has been properly loaded by typing `show -g example::*`. The output should look like the listing above.

Step 2: Create and initialize a dsp instance. After importing the Faust module you can now create an instance of the Faust signal processor using the `newinit` routine, and assign it to a Pure variable as follows:

```
> let dsp = example::newinit 44100;
```

Note that the constant 44100 denotes the desired sample rate in Hz. This can be an arbitrary integer value, which is available in the Faust program by means of the `SR` variable. It's completely up to the dsp whether it actually uses this value in some way (our example doesn't, but we need to specify a value anyway).

The dsp is now fully initialized and we can use it to compute some samples. But before we can do this, we'll need to know how many channels of audio data the dsp consumes and produces, and which control variables it provides. This information can be extracted with the `info` function, and be assigned to some Pure variables as follows:

```
> let k,l,ui = example::info dsp;
```

(We'll have a closer look at the contents of the `ui` variable below.)

In a similar fashion, the `meta` function provides some "metadata" about the Faust dsp, as a list of `key=>val` string pairs. This is static data which doesn't belong to any particular dsp instance, so it can be extracted without actually creating an instance. In our case the metadata will be empty, since we didn't supply any in the Faust program. If needed, we can add some metadata as follows:

```
declare descr   "Faust Hello World";
declare author  "Faust Guru";
declare version "1.0";
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
```

If we now reload the Faust dsp, we'll get:

```
> test::meta;
["descr"=>"Faust Hello World","author"=>"Faust Guru","version"=>"1.0"]
```

Step 3: Prepare input and output buffers. Pure's Faust interface allows you to pass Pure double matrices as sample buffers, which makes this step quite convenient. For given numbers k and l of input and output channels, respectively, we'll need a k x n matrix for the input and a l x n matrix for the output, where n is the desired block size (the number of samples to be processed per channel in one go). Note that the matrices have one row per input or output channel. Here's how we can create some suitable input and output matrices using a Pure matrix comprehension and the `dmatrix` function available in the standard library:

```
> let n = 10; // the block size
> let in  = {i*10.0+j | i = 1..k; j = 1..n};
> let out = dmatrix (l,n);
```

In our example, k=2 and l=1, thus we obtain the following matrices:

```
> in;
{11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0;
21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0,30.0}
> out;
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

Step 4: Apply the dsp to compute some samples. With the `in` and `out` matrices as given above, we can now apply the dsp by invoking its `compute` routine:

```
> example::compute dsp n in out;
```

This takes the input samples specified in the `in` matrix and stores the resulting output in the `out` matrix. The output matrix now looks as follows:

```
> out;
{9.6,10.2,10.8,11.4,12.0,12.6,13.2,13.8,14.4,15.0}
```

Note that the `compute` routine also modifies the internal state of the dsp instance so that a subsequent call will continue with the output stream where the previous call left off. (This isn't relevant in this specific example, but in general a Faust dsp may contain delays and similar constructions which need a memory of past samples to be maintained between different invocations of `compute`.) Thus we can now just keep on calling `compute` (possibly with different `in` buffers) to compute as much of the output signal as we need.

Step 5: Inspecting and modifying control variables. Recall that our sample dsp also has a Faust control variable `gain` which lets us change the amplification of the output signal. We've already assigned the corresponding information to the `ui` variable, let's have a look at it now:

```
> ui;
vgroup [] ("test",[nentry #<pointer 0x1611f00> [] ("gain",0.3,0.0,10.0,0.01)])
```

In general, this data structure takes the form of a tree which corresponds to the hierarchical layout of the control groups and values in the Faust program. In this case, we just have one toplevel group containing a single `gain` parameter, which is represented as a Pure term containing the relevant information about the type, name, initial value, range and stepsize of the control, along with a double pointer which can be used to inspect and modify the control value. While it's possible to access this information in a direct fashion, there's also a faustui.pure module in the standard library which makes this easier. First we extract the mapping of control variable names to the corresponding double pointers as follows:

```
> using faustui;
> let ui = control_map $ controls ui; ui;
{"gain"=>#<pointer 0xd81820>}
```

The result is a record value indexed by control names, thus the pointer which belongs to our `gain` control can now be obtained with `ui!"gain"`. The faustui.pure module also provides convenience functions to inspect a control and change its value:

```
> let gain = ui!"gain";
> get_control gain;
0.3
> put_control gain 1.0;
()
> get_control gain;
1.0
```

Let's rerun `compute` to get another block of samples from the same input data, using the new `gain` value:

```
> example::compute dsp n in out;
> out;
{32.0,34.0,36.0,38.0,40.0,42.0,44.0,46.0,48.0,50.0}
```

Faust also allows metadata to be attached to individual controls and control groups, which is available in the same form of a list of `key=>val` string pairs that we have seen already with the `meta` operation. This metadata is used to provide auxiliary information about a control to specific applications. It's completely up to the application how to interpret this metadata. Typical examples are style hints about GUI renderings of a control, or the assignment of external "MIDI" controllers. (**MIDI** is the "Musical Instruments Digital Interface", a standardized hardware and software interface for electronic music instruments and other digital multimedia equipment.)

In our example these metadata lists are all empty. Control metadata is specified in a Faust program in the labels of the controls using the syntax `[key:val]`, please see the Faust documentation for details. For instance, if we'd like to assign MIDI controller 7 (usually the "volume controller" on MIDI keyboards) to our `gain` control, this might be done as follows:

```
gain = nentry("gain [midi:ctrl 7]", 0.3, 0, 10, 0.01);
```

After reloading the dsp and creating a new instance, this metadata is available in the `ui` structure and can be extracted with the `control_meta` function of the faustui module as follows:

```
> let dsp = test::newinit SR;
> let k,l,ui = example::info dsp;
> controls ui!0;
nentry #<pointer 0x1c97070> ["midi"=>"ctrl 7"] ("gain",0.3,0.0,10.0,0.01)
> control_meta ans;
["midi"=>"ctrl 7"]
```

As you can see, all these steps are rather straightforward. Of course, in a real program we would probably run `compute` in a loop which reads some samples from an audio device or sound file, applies the dsp, and writes back the resulting samples to another audio device or file. We might also have to process MIDI controller input and change the control variables accordingly. This can all be done quite easily using the appropriate addon modules available on the Pure website.

We barely scratched the surface here, but it should be apparent that the programming techniques sketched out in this section open the door to the realm of sophisticated multimedia

and signal processing applications. More Faust-related examples can be found in the Pure distribution. Also, have a look at the *pd-pure* and *pd-faust* packages to see how these facilities can be used in Pd modules written in Pure.

1.11 Interactive Usage

In interactive mode, the interpreter reads definitions and expressions and processes them as usual. You can use the `-i` option to force interactive mode when invoking the interpreter with some script files. Additional scripts can be loaded interactively using either a `using` declaration or the interactive `run` command (see the description of the `run` command below for the differences between these). Or you can just start typing away, entering your own definitions and expressions to be evaluated.

The input language is mostly the same as for source scripts, and hence individual definitions and expressions must be terminated with a semicolon before they are processed. For instance, here is a simple interaction which defines the factorial and then uses that definition in some evaluations. Input lines begin with "> ", which is the interpreter's default command prompt:

```
> fact 1 = 1;
> fact n = n*fact (n-1) if n>1;
> let x = fact 10; x;
3628800
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

As indicated, in interactive mode the normal forms of toplevel expressions are printed after each expression is entered. This is also commonly known as the **read-eval-print loop**. Normal form expressions are usually printed in the same form as you'd enter them. However, there are a few special kinds of objects like anonymous closures, thunks ("lazy" values to be evaluated when needed) and pointers which don't have a textual representation in the Pure syntax and will be printed in the format #<*object description*> by default. It is also possible to override the print representation of any kind of expression by means of the `__show__` function, see Pretty-Printing below for details.

A number of other special features of Pure's command line interface are discussed in the following subsections.

1.11.1 Command Syntax

Besides Pure definitions and expressions, the interpreter also understands a number of special interactive commands for performing basic maintenance tasks, such as loading source scripts, exiting and restarting the interpreter, changing the working directory, escaping to the shell, getting help and displaying definitions. In contrast to the normal input language, the command language is line-oriented; it consists of special command words to be typed at the beginning of an input line, which may be followed by some parameters as required by the command. The command language is intended solely for interactive purposes and

thus doesn't offer any programming facilities of its own. However, it can be extended with user-defined commands implemented as ordinary Pure functions; this is described in the User-Defined Commands section below.

In fact, as of Pure 0.56 the interpreter actually provides two slightly different command syntaxes, which we'll refer to as "default" and "escape mode". The manual assumes that you're running the interpreter in its traditional **default mode** where interactive commands are typed simply as they are shown in the following subsections, with the command word at the very beginning of the line. However, this mode has its pitfalls, especially for beginners. As most of the commands look just like ordinary identifiers, you may run into situations where the beginning of an expression or definition to be typed at the prompt can be mistaken for a command word. In such cases the default mode requires that you insert one or more spaces at the beginning of the line, so that the interpreter reads the line as normal Pure code. Unfortunately, it's much too easy to forget this if you're not familiar with the command language.

Therefore there is an alternative **escape mode** available which handles special command input more like some other popular programming language interpreters. In escape mode *all* interactive commands have to be escaped by prefixing them with a special character at the very beginning of the line. The command itself must follow the prefix character, without any intervening whitespace. Any line not prefixed with the prefix character will then be considered normal Pure code. This mode can be enabled with the `--escape` option, which takes the desired prefix character as an argument, or you can just set the `PURE_ESCAPE` variable in your environment to enable escape mode by default.

For example, to set the escape character to ':' you'll invoke the interpreter as follows:

```
$ pure --escape=':'
```

Alternatively, you could also set the `PURE_ESCAPE` environment variable like this (using Bourne shell syntax):

```
$ export PURE_ESCAPE=':'
```

Note that specifying the prefix character with the `--escape` option overrides the value of the environment variable, and only the initial character in the value of `--escape` or `PURE_ESCAPE` will be used. If the specified value is empty, the interpreter reverts to the default mode. The following prefix characters can be used: `!$%&*,:<>@\|`. Note that these all belong to 7 bit ASCII, and only some of the ASCII punctuation characters are permitted in order to prevent conflicts with ordinary Pure code. In any case, all of these characters *can* also occur in ordinary Pure code, so you should use a prefix that you aren't likely to type at the beginning of a line in your usual coding style.

Many Pure programmers prefer escape mode, and in fact we recommend it for Pure novices even though it's not the default (yet). Others may prefer default mode because it's less effort to type. For the manual we stick to the default mode syntax. This means that if you're running the interpreter in escape mode then you'll have to do the necessary translation of the command syntax yourself. For instance, if the manual tells you to type the following command,

```
> show foldl
```

and you are using ':' as the command prefix, then you will have to type this in escape mode instead:

```
> :show foldl
```

Note that in this case '!' continues to serve as a shell escape:

```
> ! find . '*.pure'
```

This will not work, however, if you use '!' as your command prefix. In this case you will have to type *two* exclamation marks instead (the same caveat applies if you escape a shell command in the debugger, cf. Debugging):

```
> !! find . '*.pure'
```

This should be rather straightforward, so in the following we just use the default mode command syntax throughout without further notice.

Note: Escape mode only applies to the interactive command line. It doesn't affect the `evalcmd` function in any way, so interactive commands in the string argument of `evalcmd` are always specified without the escape character prefix no matter which mode the interpreter is running in.

1.11.2 Online Help

Online help is available in the interpreter with the interactive `help` command, which gives you access to all the available documentation in html format; this includes the present manual, the *Pure Library Manual*, as well as all manuals of the addon modules available from the Pure website.

You need to have a html browser installed to make this work. By default, the `help` command uses **w3m**, but you can change this by setting either the `PURE_HELP` or the `BROWSER` environment variable accordingly.

When invoked without arguments, the `help` command displays an overview of the available documentation, from which you can follow the links to the provided manuals:

```
> help
```

(If the interpreter gives you an error message when you do this then you haven't installed the documentation yet. The complete set of manuals is provided as a separate package at the Pure website, please see the Pure installation instructions for details.)

The `help` command also accepts a parameter which lets you specify a search term which is looked up in the global index, e.g.:

```
> help foldl
```

Besides Pure functions, macros, variables and constants described in the manual you can also look up program options and environment variables, e.g.:

```
> help -x
> help pure-gen -x
> help PURE_STACK
```

(Note that you can specify the program name to disambiguate between options for different utilities, such as the `-x` option which is accepted both by the Pure interpreter and the `pure-gen` program.)

If the search term doesn't appear in the index, it is assumed to be a topic (a link target, usually a section title) in the Pure manual. Note that the docutils tools used to generate the html source of the Pure documentation mangle the section titles so that they are in lowercase and blanks are replaced with hyphens. So to look up the present section in this manual you'd have to type:

```
> help online-help
```

The help files are in html format and located in the docs subdirectory of the Pure library directory (i.e., /usr/local/lib/pure/docs by default). You can look up topics in any of the help files with a command like the following:

```
> help pure-gsl#matrices
```

Here `pure-gsl` is the basename of the help file (library path and .html suffix are supplied automatically), and `matrices` is a link target in that document. To just read the pure-gsl.html file without specifying a target, type the following:

```
> help pure-gsl#
```

(Note that just `help pure-gsl` won't work, since it would look for a search term in the index or a topic in the Pure manual.)

Last but not least, you can also point the help browser to any html document (either a local file or some website) denoted by a proper URL, provided that your browser program can handle these. For instance:

```
> help file:mydoc.html#foo
> help http://pure-lang.googlecode.com
```

1.11.3 Interactive Commands

The following built-in commands are always understood by the interpreter. (In addition, you can define your own commands for frequently-used operations; see User-Defined Commands below.)

`! command`
Shell escape.

`break [symbol ...]`
Sets breakpoints on the given function or operator symbols. All symbols must be specified in fully qualified form, see the remarks below. If invoked without arguments, prints all currently defined breakpoints. This requires that the interpreter was invoked with the `-g` option to enable debugging support. See Debugging below for details.

`bt`
Prints a full backtrace of the call sequence of the most recent evaluation, if that evaluation ended with an unhandled exception. This requires that the interpreter was invoked with the `-g` option to enable debugging support. See Debugging below for details.

`cd dir`
Change the current working dir.

`clear [option ...] [symbol ...]`
Purge the definitions of the given symbols (functions, macros, constants or global variables). All symbols must be specified in fully qualified form, see the remarks below. If invoked as `clear ans`, clears the `ans` value (see Last Result below). When invoked without any arguments, `clear` purges all definitions at the current interactive "level" (after confirmation) and returns you to the previous level, if any. (It might be a good idea to first check your current definitions with `show` or back them up with `dump` before you do that.) The desired level can be specified with the `-t` option. See the description of the `save` command and Definition Levels below for further details. A description of the common options accepted by the `clear`, `dump` and `show` commands can be found in Specifying Symbol Selections below.

`del [-b|-m|-t] [symbol ...]`
Deletes breakpoints and tracepoints on the given function or operator symbols. If the `-b` option is specified then only breakpoints are deleted; similarly, `del -t` only deletes tracepoints. If none of these are specified then both breakpoints and tracepoints are deleted. All symbols must be specified in fully qualified form, see the remarks below. If invoked without non-option arguments, `del` clears *all* currently defined breakpoints and/or tracepoints (after confirmation); see Debugging below for details.

The `-m` option works similarly to `-t`, but deletes macro rather than function tracepoints, see the description of the `trace` command below.

`dump [-n filename] [option ...] [symbol ...]`
Dump a snapshot of the current function, macro, constant and variable definitions in Pure syntax to a text file. All symbols must be specified in fully qualified form, see the remarks below. This works similar to the `show` command (see below), but writes the definitions to a file. The default output file is .pure in the current directory, which is then reloaded automatically the next time the interpreter starts up in interactive mode in the same directory. This provides a quick-and-dirty way to save an interactive session and have it restored later, but note that this isn't perfect. In particular, declarations of `extern` symbols won't be saved unless they're specified explicitly, and some objects

like closures, thunks and pointers don't have a textual representation from which they could be reconstructed. To handle these, you'll probably have to prepare a corresponding .purerc file yourself, see Interactive Startup below.

A different filename can be specified with the `-n` option, which expects the name of the script to be written in the next argument, e.g: `dump -n myscript.pure`. You can then edit that file and use it as a starting point for an ordinary script or a .purerc file, or you can just run the file with the `run` command (see below) to restore the definitions in a subsequent interpreter session.

`help [topic]`
Display online documentation. If a topic is given, it is looked up in the index. Alternatively, you can also specify a link target in any of the installed help files, or any other html document denoted by a proper URL. Please see Online Help above for details.

`ls [args]`
List files (shell **ls** command).

`mem`
Print current memory usage. This reports the number of expression cells currently in use by the program, along with the size of the freelist (the number of allocated but currently unused expression cells). Note that the actual size of the expression storage may be somewhat larger than this, since the runtime always allocates expression memory in bigger chunks. Also, this figure does not reflect other heap-allocated memory in use by the program, such as strings or malloc'ed pointers.

`override`
Enter "override" mode. This allows you to add equations "above" existing definitions in the source script, possibly overriding existing equations. See Definition Levels below for details.

`pwd`
Print the current working dir (shell **pwd** command).

`quit`
Exits the interpreter.

`run [-g|script]`
When invoked without arguments or with the `-g` option, `run` does a "cold" restart of the interpreter, with the scripts and options given on the interpreter's original command line. If just `-g` is specified as the argument, the interpreter is run with debugging enabled. Otherwise the interpreter is invoked without debugging support. (This overrides the corresponding option from the interpreter's command line.) This command provides a quick way to rerun the interpreter after changes in some of the loaded script files, or if you want to enable or disable debugging on the fly (which requires a restart of the interpreter). You'll also loose any definitions that you entered interactively in the interpreter, so you may want to back them up with `dump` beforehand.

When invoked with a script name as argument, `run` loads the given script file and adds its definitions to the current environment. This works more or less like a `using` clause, but only searches for the script in the current directory and places the definitions in the

script at the current temporary level, so that `clear` can be used to remove them again. Also note that namespace and pragma settings of scripts loaded with `run` stick around after loading the script. This allows you to quickly set up your environment by just running a script containing the necessary namespace declarations and compiler directives. (Alternatively, you can also use the interpreter's startup files for that purpose, see Interactive Startup below.)

`save`
Begin a new level of temporary definitions. A subsequent `clear` command (see above) will purge the definitions made since the most recent `save` command. See Definition Levels below for details.

`show [option ...] [symbol ...]`
Show the definitions of symbols in various formats. See The show Command below for details. All symbols must be specified in fully qualified form, see the remarks below. A description of the common options accepted by the `clear`, `dump` and `show` commands can be found in Specifying Symbol Selections below.

`stats [-m] [on|off]`
Enables (default) or disables "stats" mode, in which some statistics are printed after an expression has been evaluated. Invoking just `stats` or `stats on` only prints the cpu time in seconds for each evaluation. If the `-m` option is specified, memory usage is printed along with the cpu time, which indicates the maximum amount of expression memory (in terms of expression cells) used during the computation. Invoking `stats off` disables stats mode, while `stats -m off` just disables the printing of the memory usage statistics.

`trace [-a] [-m] [-r] [-s] [symbol ...]`
Sets tracepoints on the given function or operator symbols. Without the `-m` option, this works pretty much like the `break` command (see above) but only prints rule invocations and reductions without actually interrupting the evaluation; see Debugging below for details.

The `-m` option allows you to trace macro (rather than function) calls. If this option is specified, the compiler prints reduction sequences involving the given macro symbol, which is useful when debugging macros; see the Macros section for details and examples. Note that macro tracing works even if the interpreter was invoked without debugging mode.

If the `-a` option is specified, tracepoints are set on *all* global function or macro symbols, respectively (in this case the symbol arguments are ignored). This is convenient if you want to see any and all reductions performed in a computation.

Tracing can actually be performed in two different modes, *recursive* mode in which the trace is triggered by any of the active tracepoints and continues until the corresponding call is finished, or *skip* mode in which *only* calls by the active tracepoints are reported. The former is usually more helpful and is the default. The `-s` option allows you to switch to skip mode, while the `-r` option switches back to recursive mode.

Finally, if neither symbols nor any of the `-a`, `-r` and `-s` options are specified then the

currently defined tracepoints are printed. Note that, as with the `break` command, existing tracepoints can be deleted with the `del` command (see above).

`underride`
Exits "override" mode. This returns you to the normal mode of operation, where new equations are added "below" previous rules of an existing function. See Definition Levels below for details.

Note that symbols (identifiers, operators etc.) must always be specified in fully qualified form. No form of namespace lookup is performed by commands like `break`, `clear`, `show` etc. Thus the specified symbols always work the same no matter what `namespace` and `using namespace` declarations are currently in effect.

Besides the commands listed above, the interpreter also provides some special commands for the benefit of other programs such as **emacs** driving the interpreter; currently these are `completion_matches`, `help_matches` and `help_index`. These aren't supposed to be invoked directly by the user, although they may sometimes be useful to implement custom functionality, see User-Defined Commands.

1.11.4 Specifying Symbol Selections

The `clear`, `dump` and `show` commands all accept the following options for specifying a subset of symbols and definitions on which to operate. All symbols must be specified in fully qualified form. Options may be combined, thus, e.g., `show -mft` is the same as `show -m -f -t`. Some options specify optional numeric parameters; these must follow immediately behind the option character if present, as in `-t0`.

`-c` Select defined constants.

`-f` Select defined functions.

`-g` Indicates that the following symbols are actually shell glob patterns and that all matching symbols should be selected.

`-m` Select defined macros.

`-pflag` Select only private symbols if *flag* is nonzero (the default), otherwise (*flag* is zero) select only public symbols. If this option is omitted then both private and public symbols are selected.

`-tlevel` Select symbols and definitions at the given "level" of definitions and above. This is described in more detail below. Briefly, the executing program and all imported modules (including the prelude) are at level 0, while "temporary" definitions made interactively in the interpreter are at level 1 and above. Thus a level of 1 restricts the selection to all temporary definitions, whereas 0 indicates all definitions (i.e., everything, including the prelude). If *level* is omitted, it defaults to the current definitions level.

`-v` Select defined variables.

`-y` Select defined types.

In addition, the `-h` option prints a short help message describing all available options of the command at hand.

If none of the `-c`, `-f`, `-m`, `-v` and `-y` options are specified, then all kinds of symbols (constants, functions, macros, variables and types) are selected, otherwise only the specified categories will be considered.

A reasonable default is used if the `-t` option is omitted. By default, if no symbols are specified, only temporary definitions are considered, which corresponds to `-t1`. Otherwise the command applies to all corresponding definitions, no matter whether they belong to the executing program, the prelude, or some temporary level, which has the same effect as `-t0`. This default choice can be overridden by specifying the desired level explicitly.

As a special case, just `clear` (without any other options or symbol arguments) always backs out to the previous definitions level (instead of level #1). This is inconsistent with the rules set out above, but is implemented this way for convenience and backward compatibility. Thus, if you really want to delete all your temporary definitions, use `clear -t1` instead. When used in this way, the `clear` command will only remove temporary definitions; if you need to remove definitions at level #0, you must specify those symbols explicitly.

Note that `clear -g *` will have pretty much the same disastrous consequences as the Unix command `rm -rf *`, so don't do that. Also note that a macro or function symbol may well have defining equations at different levels, in which case a command like `clear -tn foo` might only affect some part of `foo`'s definition. The `dump` and `show` commands work analogously (albeit less destructively). See Definition Levels below for some examples.

1.11.5 The show Command

The `show` command can be used to obtain information about defined symbols in various formats. Besides the common selection options discussed above, this command recognizes the following additional options for specifying the content to be listed and the format to use.

`-a` Disassembles pattern matching automata. Works like the `-v4` option of the interpreter.

`-d` Disassembles LLVM IR, showing the generated LLVM assembler code of a function. Works like the `-v8` option of the interpreter.

`-e` Annotate printed definitions with lexical environment information (de Bruijn indices, subterm paths). Works like the `-v2` option of the interpreter.

`-l` Long format, prints definitions along with the summary symbol information. This implies `-s`.

`-s` Summary format, print just summary information about listed symbols.

Symbols are always listed in lexicographic order. Note that some of the options (in particular, `-a` and `-d`) may produce excessive amounts of information. By setting the `PURE_MORE` environment variable, you can specify a shell command to be used for paging, usually **more** or **less**.

For instance, to list all temporary definitions made in an interactive session, simply say:

```
> show
```

You can also list a specific symbol, no matter whether it comes from the interactive command line, the executing script or the prelude:

```
> show foldl
foldl f a x::matrix = foldl f a (list x);
foldl f a s::string = foldl f a (chars s);
foldl f a [] = a;
foldl f a (x:xs) = foldl f (f a x) xs;
```

Wildcards can be used with the `-g` option, which is useful if you want to print an entire family of related functions, e.g.:

```
> show -g foldl*
foldl f a x::matrix = foldl f a (list x);
foldl f a s::string = foldl f a (chars s);
foldl f a [] = a;
foldl f a (x:xs) = foldl f (f a x) xs;
foldl1 f x::matrix = foldl1 f (list x);
foldl1 f s::string = foldl1 f (chars s);
foldl1 f (x:xs) = foldl f x xs;
```

Or you can just specify multiple symbols as follows (this also works with multiple glob patterns when you add the `-g` option):

```
> show min max
max x y = if x>=y then x else y;
min x y = if x<=y then x else y;
```

You can also select symbols by category. E.g., the following command shows summary information about all the variable symbols along with their current values (using the "long" format):

```
> show -lvg *
argc       var  argc = 0;
argv       var  argv = [];
compiling  var  compiling = 0;
sysinfo    var  sysinfo = "x86_64-unknown-linux-gnu";
version    var  version = "0.56";
5 variables
```

Or you can list just private symbols of the namespace `foo`, as follows:

```
> show -pg foo::*
```

The following command will list each and every symbol that's currently defined (instead of `-g *` you can also use the `-t0` option):

```
> show -g *
```

This usually produces a lot of output and is rarely needed, unless you'd like to browse through an entire program including all library imports. (In that case you might consider

to use the `dump` command instead, which writes the definitions to a file which can then be loaded into a text editor for easier viewing. This may occasionally be useful for debugging purposes.)

The `show` command also has the following alternate forms which are used for special purposes:

- `show interface` lists the actual `type` rules for an interface type. This is useful if you want to verify which patterns will be matched by an interface type, see Interface Types for details. For instance:

```
> interface stack with
>   push xs::stack x;
>   pop xs::stack;
>   top xs::stack;
> end;
> push xs@[] x |
> push xs@(_:_) x = x:xs;
> pop (x:xs) = xs;
> top (x:xs) = x;
> show interface stack
type stack xs@(_:_);
> pop [] = throw "empty stack";
> top [] = throw "empty stack";
> show interface stack
type stack xs@[];
type stack xs@(_:_);
```

- `show namespace` lists the current and search namespaces, while `show namespaces` lists all declared namespaces. These come in handy if you have forgotten what namespaces are currently active and which other namespaces are available in your program. For instance:

```
> show namespace
> show namespaces
namespace C;
namespace matrix;
> using namespace C;
> namespace my;
> show namespace
namespace my;
using namespace C;
```

1.11.6 Definition Levels

To help with incremental development, the interpreter offers some commands to manipulate the current set of definitions interactively. To these ends, definitions are organized into different subsets called **levels**. As already mentioned, the prelude, as well as other source programs specified when invoking the interpreter, are always at level 0, while the interactive environment starts at level 1. Each `save` command introduces a new temporary level, and

each subsequent `clear` command (without any arguments) "pops" the definitions on the current level and returns you to the previous one (if any). This gives you a "stack" of temporary environments which enables you to "plug and play" in a (more or less) safe fashion, without affecting the rest of your program.

For all practical purposes, this stack is unlimited, so that you can create as many levels as you like. However, this facility also has its limitations. The interpreter doesn't really keep a full history of everything you entered interactively, it only records the level a variable, constant, and function or macro rule belongs to so that the corresponding definitions can be removed again when the level is popped. On the other hand, intermediate changes in variable values are not recorded anywhere and cannot be undone. Moreover, global declarations (which encompasses `using` clauses, `extern` declarations and special symbol declarations) always apply to all levels, so they can't be undone either.

That said, the temporary levels can still be pretty useful when you're playing around with the interpreter. Here's a little example which shows how to use `clear` to quickly get rid of a definition that you entered interactively:

```
> foo (x:xs) = x+foo xs;
> foo [] = 0;
> show
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
> clear
This will clear all temporary definitions at level #1.
Continue (y/n)? y
> show
> foo (1..10);
foo [1,2,3,4,5,6,7,8,9,10]
```

We've seen already that normally, if you enter a sequence of equations, they will be recorded in the order in which they were written. However, it is also possible to override definitions in lower levels with the `override` command:

```
> foo (x:xs) = x+foo xs;
> foo [] = 0;
> show
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
> save
save: now at temporary definitions level #2
> override
> foo (x:xs) = x*foo xs;
> show
foo (x:xs) = x*foo xs;
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
```

```
warning: rule never reduced: foo (x:xs) = x+foo xs;
0
```

Note that the equation `foo (x:xs) = x*foo xs` was inserted before the previous rule `foo (x:xs) = x+foo xs`, which is at level #1. (The latter equation is now "shadowed" by the rule we just entered, hence the compiler warns us that this rule can't be reduced any more.)

Even in override mode, new definitions will be added after other definitions at the *current* level. This allows us to just continue adding more high-priority definitions overriding lower-priority ones:

```
> foo [] = 1;
> show
foo (x:xs) = x*foo xs;
foo [] = 1;
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
warning: rule never reduced: foo (x:xs) = x+foo xs;
warning: rule never reduced: foo [] = 0;
3628800
```

Again, the new equation was inserted above the existing lower-priority rules, but below our previous equation `foo (x:xs) = x*foo xs` entered at the same level. As you can see, we have now effectively replaced our original definition of `foo` with a version that calculates list products instead of sums, but of course we can easily go back one level to restore the previous definition:

```
> clear
This will clear all temporary definitions at level #2.
Continue (y/n)? y
clear: now at temporary definitions level #1
clear: override mode is on
> show
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
```

Note that `clear` reminded us that override mode is still enabled (`save` will do the same if override mode is on while pushing a new definitions level). To turn it off again, use the `underride` command. This will revert to the normal behaviour of adding new equations below existing ones:

```
> underride
```

It's also possible to use `clear` to back out multiple levels at once, if you specify the target level to be cleared with the -t option. For instance:

```
> save
save: now at temporary definitions level #2
> let bar = 99;
```

```
> show
let bar = 99;
foo (x:xs) = x+foo xs;
foo [] = 0;
> // this scraps all our scribblings!
> clear -t1
This will clear all temporary definitions at level #1 and above.
Continue (y/n)? y
clear: now at temporary definitions level #1
> show
>
```

The facilities described above are also available to Pure programs, as the `save` and `clear` commands can also be executed under program control using the `evalcmd` primitive. Conversely, the library provides its own functions for inspecting and manipulating the source program, which may also be useful in custom command definitions; see the *Pure Library Manual* for details.

1.11.7 Debugging

The interpreter provides a simple but reasonably convenient symbolic debugging facility when running interactively. To make this work, you have to specify the `-g` option when invoking the interpreter (`pure -g`). If you're already at the interpreter's command line, you can also use the `run -g` command to enable the debugger. The `-g` option disables tail call optimization (see Stack Size and Tail Recursion) to make it easier to debug programs. It also causes special debugging code to be generated which will make your program run *much* slower. Therefore the `-g` option should only be used if you actually need the debugger.

One common use of the debugger is "post mortem" debugging after an evaluation ended with an unhandled exception. In such a case, the `bt` command of the interpreter prints a backtrace of the call sequence which caused the exception. Note that this only works if debugging mode was enabled. For instance:

```
> [1,2]!3;
<stdin>, line 2: unhandled exception 'out_of_bounds' while evaluating '[1,2]!3'
> bt
   [1] (!): (x:xs)!n::int = xs!(n-1) if n>0;
     n = 3; x = 1; xs = [2]
   [2] (!): (x:xs)!n::int = xs!(n-1) if n>0;
     n = 2; x = 2; xs = []
   [3] (!): []!n::int = throw out_of_bounds;
     n = 1
>> [4] throw: extern void pure_throw(expr*) = throw;
     x1 = out_of_bounds
```

The last call, which is also marked with the `>>` symbol, is the call that raised the exception. The format is similar to the `p` command of the debugger, see below, but `bt` always prints a full backtrace. (As with the `show` command of the interpreter, you can set the `PURE_MORE` environment variable to pipe the output through the corresponding command, or use `evalcmd`

to capture the output of `bt` in a string.)

The debugger can also be used interactively. To these ends, you can set breakpoints on functions with the `break` command. The debugger then gets invoked as soon as a rule for one of the given functions is executed. Example:

```
> fact n::int = if n>0 then n*fact (n-1) else 1;
> break fact
> fact 1;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
(Type 'h' for help.)
:
** [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
:
++ [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
     --> 1
** [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
:
++ [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
     --> 1
++ [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
     --> 1
1
```

Lines beginning with ** indicate that the evaluation was interrupted to show the rule (or external) which is currently being considered, along with the current depth of the call stack, the invoked function and the values of parameters and other local variables in the current lexical environment. In contrast, the prefix ++ denotes reductions which were actually performed during the evaluation and the results that were returned by the function call (printed as `--> return value`).

Sometimes you might also see funny symbols like `#<closure>`, `#<case>` or `#<when>` instead of the function name. These indicate lambdas and the special variable-binding environments, which are all implemented as anonymous closures in Pure. Also note that the debugger doesn't know about the argument names of external functions (which are optional in Pure and not recorded anywhere), so it will display the generic names `x1`, `x2` etc. instead.

At the debugger prompt ':' you can enter various special debugger commands, or just keep on hitting the carriage return key to walk through an evaluation step by step, as we did in the example above. (Command line editing works as usual at the debugger prompt, if it is enabled.) The usual commands are provided to walk through an evaluation, print and navigate the call stack, step over the current call, or continue the evaluation unattended until you hit another breakpoint. If you know other source level debuggers like **gdb** then you should feel right at home. You can type `h` at the debugger prompt to print the following list:

```
: h
Debugger commands:
a       auto: step through the entire program, run unattended
c [f]   continue until next breakpoint, or given function f
h       help: print this list
n       next step: step over reduction
p [n]   print rule stack (n = number of frames)
r       run: finish evaluation without debugger
s       single step: step into reduction
t, b    move to the top or bottom of the rule stack
u, d    move up or down one level in the rule stack
x       exit the interpreter (after confirmation)
.       reprint current rule
! cmd   execute interpreter command
? expr  evaluate expression
<cr>    single step (same as 's')
<eof>   step through program, run unattended (same as 'a')
```

Note: If you specified an `--escape` prefix other than '`!`' (cf. Command Syntax), that prefix will be used to execute interpreter commands instead, see below. The help message will tell you which command prefix is in effect.

The command syntax is very simple. Besides the commands listed above you can also enter comment lines (`// comment text`) which will just be ignored. Extra arguments on commands which don't expect any will generally be ignored as well. The single letter commands all have to be separated from any additional parameters with whitespace, whereas the '`!`', '`?`' and '`.`' commands count as word delimiters and can thus be followed immediately by an argument. For convenience, the '`?`' command can also be omitted if the expression to be evaluated doesn't start with a single letter or one of the special punctuation commands.

The debugger can be exited or suspended in the following ways:

- You can type `c` to continue the evaluation until the next breakpoint, or `c foo` in order to proceed until the debugger hits an invocation of the function `foo`.
- You can type `r` to run the rest of the evaluation without the debugger.
- The `a` ("auto") command single-steps through the rest of the evaluation, running unattended. This command can also be entered by just hitting the end-of-file key (`Ctrl-d` on Unix systems) at the debugger prompt.
- You can also type `x` to exit from the debugger *and* the interpreter immediately (after confirmation).

In addition, you can use the `!` command (or whatever command prefix has been set with the `--escape` option) to run any interpreter command while in the debugger. For instance:

```
: !ls
```

This is particularly useful to invoke the `break` and `del` commands to change breakpoints. Note that you can actually escape any valid input to the interpreter that way, not just the

special interactive commands. However, you shouldn't try to modify the program while you're debugging it. This may work in some cases, but will have nasty consequences if you happen to change a function which is currently being executed.

The interpreter's shell escape can also be used from the debugger. In default mode or when using `!` as the *`--escape`* prefix, you'll have to escape shell commands with `!!`, otherwise a single `!` suffices.

At the debugger prompt, you can use the `u` ("up"), `d` ("down"), `t` ("top") and `b` ("bottom") commands to move around on the current call stack. The `p` command prints a range of the call stack centered around the currently selected stack frame, which is indicated with the `>>` tag, whereas `**` denotes the current bottom of the stack (which is the rule to be executed with the single step command). The `p` command can also be followed by a numeric argument which indicates the number of stack frames to be printed (this will then become the default for subsequent invocations of `p`). The `n` command steps over the call selected with the stack navigation commands. For instance:

```
> fact 3;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 3
: c *
** [4] (*): x::int*y::int = x*y;
     x = 1; y = 1
: p
   [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 3
   [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 2
   [3] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
** [4] (*): x::int*y::int = x*y;
     x = 1; y = 1
: u
>> [3] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
: u
>> [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 2
: p
   [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 3
>> [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 2
   [3] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
** [4] (*): x::int*y::int = x*y;
     x = 1; y = 1
: n
++ [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 2
     --> 2
** [2] (*): x::int*y::int = x*y;
```

```
     x = 3; y = 2
:
```

If you ever get lost, you can reprint the current rule with the '`.`' command:

```
: .
** [2] (*): x::int*y::int = x*y;
     x = 3; y = 2
```

Another useful feature is the `?` command which lets you evaluate any Pure expression, with the local variables of the current rule bound to their corresponding values. Like the `n` command, `?` applies to the current stack frame as selected with the stack navigation commands. The expression must be entered on a single line, and the trailing semicolon is optional. For instance:

```
> fact 3;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 3
: c *
** [4] (*): x::int*y::int = x*y;
     x = 1; y = 1
: ?x+y
2
: u
>> [3] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
: n>0, fact n
1,1
```

A third use of the debugger is to trace function calls. For that the interpreter provides the `trace` command which works similarly to `break`, but sets so-called "tracepoints" which only print rule invocations and reductions instead of actually interrupting the evaluation. For instance, assuming the same example as above, let's first remove the breakpoint on `fact` (using the `del` command) and then set it as a tracepoint instead:

```
> del fact
> trace fact
> fact 1;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
** [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
++ [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
     --> 1
** [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
++ [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
     --> 1
++ [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
```

```
     --> 1
1
```

The `break` and `trace` commands can also be used in concert if you want to debug some functions while only tracing others.

Note that the `trace` command can actually be run in two different modes: *recursive* mode in which the trace is triggered by any of the active tracepoints and continues until the corresponding call is finished, or *skip* mode in which *only* calls by the active tracepoints are reported. The former is the default and is often preferable, because it gives you a complete transcript of the reductions performed during a global function call, including reductions of local and anonymous function applications.

If you don't need that much detail, you can also switch to skip mode by invoking the `trace` command with the `-s` option. This allows you to see a quick summary of the computation which only shows reductions by rules directly involving the active tracepoints. For instance:

```
> trace -s
> fact 1;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
** [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
++ [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
     --> 1
++ [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
     --> 1
1
```

Moreover, the `trace` command can also be invoked with the `-a` option to trace all function calls, which is convenient to quickly obtain a full transcript of a reduction sequence. The same options also work in an analogous fashion with macro calls, see the Macros section for some examples.

The current sets of breakpoints and tracepoints can be changed with the `break`, `trace` and `del` commands, as shown above, and just `break` or `trace` without any arguments lists the currently defined breakpoints or tracepoints, respectively. Please see Interactive Commands above for details. Also note that these are really interpreter commands, so to invoke them in the debugger you have to escape them with the `!` command (or whatever other `--escape` prefix you specified).

The debugger can also be triggered programmatically with the built-in parameter-less functions `break` and `trace`. This gives you much better control over the precise location and the conditions under which the debugger should be invoked. Just place a call to `break` or `trace` near the point where you'd like to start debugging or tracing; this can be done either with the sequencing operator '`$$`' or with a `when` clause. The debugger will then be invoked at the next opportunity (usually when a function is called or a reduction is completed). For instance:

```
> fact n::int = if n>0 then break $$ n*fact (n-1) else 1;
> fact 10;
++ [2] break: extern void pure_break() = break;
     --> ()
** [2] fact: fact n::int = if n>0 then break$$n*fact (n-1) else 1;
     n = 9
:
```

Here the debugger is invoked right after the call to `break`, when the `n*fact (n-1)` expression in the `then` branch is about to be evaluated. The debugger thus stops at the recursive invocation of `fact 9`. Tracing works in a similar fashion, using `trace` in lieu of `break`, and continues until the current stack frame is exited. One major advantage of this method is that it is possible to invoke `break` or `trace` only under certain conditions, so that you can focus on interesting "events" during evaluation, which can make debugging much less tedious. In our example, in order to stop when `n` becomes `1`, we might invoke `break` as follows:

```
> fact n::int = if n>0 then n>1||break $$ n*fact (n-1) else 1;
> fact 3;
++ [4] break: extern void pure_break() = break;
     --> ()
** [4] fact: fact n::int = if n>0 then n>1||break$$n*fact (n-1) else 1;
     n = 0
: p
   [1] fact: fact n::int = if n>0 then n>1||break$$n*fact (n-1) else 1;
     n = 3
   [2] fact: fact n::int = if n>0 then n>1||break$$n*fact (n-1) else 1;
     n = 2
   [3] fact: fact n::int = if n>0 then n>1||break$$n*fact (n-1) else 1;
     n = 1
** [4] fact: fact n::int = if n>0 then n>1||break$$n*fact (n-1) else 1;
     n = 0
:
```

1.11.8 Last Result

Another convenience for interactive usage is the `ans` function, which retrieves the most recent result printed in interactive mode. For instance:

```
> fact n = if n<=1 then 1 else n*fact (n-1);
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
> scanl (+) 0 ans;
[0,1,3,9,33,153,873,5913,46233,409113,4037913]
```

Note that `ans` is just an ordinary function, defined in the prelude, not a special command. However, there is a special `clear ans` command which purges the `ans` value. This is useful, e.g., if you got a huge result which you want to erase from memory before starting the next computation.

```
> clear ans
> ans;
ans
```

1.11.9 Pretty-Printing

The interpreter provides the following "hook" to override the print representations of expressions. This works in a fashion similar to Haskell's `show` function.

__show__ x
: The programmer may define this function to supply custom print representations for certain expressions.

`__show__` is just an ordinary Pure function expected to return a string with the desired custom representation of a normal form value given as the function's single argument. The interpreter prints the strings returned by `__show__` just as they are. It will not check whether they conform to Pure syntax and/or semantics, or modify them in any way. Also, the library doesn't define this function anywhere, so you are free to add any rules that you want.

Custom print representations are most useful for interactive purposes, if you're not happy with the default print syntax of some kinds of objects. One particularly useful application of `__show__` is to change the format of numeric values. Here are some examples:

```
> using system;
> __show__ x::double = sprintf "%0.6f" x;
> 1/7;
0.142857
> __show__ x::int = sprintf "0x%0x" x;
> 1786;
0x6fa
> using math;
> __show__ (x::double +: y::double) = sprintf "%0.6f+%0.6fi" (x,y);
> cis (-pi/2);
0.000000+-1.000000i
```

The prelude function `str`, which returns the print representation of any Pure expression, calls `__show__` as well:

```
> str (1/7);
"0.142857"
```

Conversely, you can call the `str` function from `__show__`, but in this case it always returns the default representation of an expression. This prevents the expression printer from going recursive, and allows you to define your custom representation in terms of the default one. E.g., the following rule removes the `L` suffixes from bigint values:

```
> __show__ x::bigint = init (str x);
> fact n = foldl (*) 1L (1..n);
> fact 30;
265252859812191058636308480000000
```

Of course, your definition of `__show__` can also call `__show__` itself recursively to determine the custom representation of an object.

One case which needs special consideration are thunks (futures). The printer will *never* use `__show__` for those, to prevent them from being forced inadvertently. In fact, you *can* use `__show__` to define custom representations for thunks, but only in the context of a rule for other kinds of objects, such as lists. For instance:

```
> nonfix ...;
> __show__ (x:xs) = str (x:...) if thunkp xs;
> 1:2:(3..inf);
1:2:3:...
```

Another case which needs special consideration are numeric matrices. For efficiency, the expression printer will always use the default representation for these, unless you override the representation of the matrix as a whole. E.g., the following rule for double matrices mimics Octave's default output format (for the sake of simplicity, this isn't perfect, but you get the idea):

```
> __show__ x::matrix =
>   strcat [printd j (x!(i,j))|i=0..n-1; j=0..m-1] + "\n"
> with printd 0 = sprintf "\n%10.5f"; printd _ = sprintf "%10.5f" end
> when n,m = dim x end if dmatrixp x;
> {1.0,1/2;1/3,4.0};
   1.00000   0.50000
   0.33333   4.00000
```

Finally, by just purging the definition of the `__show__` function you can easily go back to the standard print syntax:

```
> clear __show__
> 1/7; 1786; cis (-pi/2);
0.142857142857143
1786
6.12303176911189e-17+:-1.0
```

Note that if you have a set of definitions for the `__show__` function which should always be loaded at startup, you can put them into the interpreter's interactive startup files, see Interactive Startup below.

1.11.10 User-Defined Commands

It is possible to extend the interpreter with your own interactive commands. To these ends, all you have to do is provide some corresponding public function definitions in the special `__cmd__` namespace (cf. Namespaces). These definitions are typically placed in one of the interpreter's startup files (see Interactive Startup below) so that they are always available when running the interpreter interactively.

A command function is invoked with one string argument which contains the rest of the command line (with leading and trailing whitespace stripped off). It may return a string

result which is printed on standard output (appending a newline if needed). Thus a simple command which just prints its arguments as is can be implemented as follows:

```
> namespace __cmd__;
> echo s = s;
> echo Hello, world!
Hello, world!
```

You can split arguments and do any required processing of the arguments with the usual string processing functions. For instance, let's change our `echo` command so that it prints each whitespace-delimited token on a line of its own:

```
> clear __cmd__::echo
> echo s = join "\n" args when
>   args = [a | a = split " " s; ~null a];
> end;
> echo Hello, world!
Hello,
world!
```

A command function may in fact return any kind of value. However, only string results are printed by the interpreter, other results are silently ignored. Thus we might implement the `echo` command in a direct fashion, using the C `puts` function:

```
> clear __cmd__::echo
> private extern int puts(char*);
> echo s = puts s;
> echo Hello, world!
Hello, world!
```

Note that we declared `puts` as a private symbol here. In general, the interpreter only exposes public functions in the `__cmd__` namespace as commands, private symbols are hidden. On the other hand, we might also just expose the external function `puts` itself under the (public) alias `echo`, so here's yet another possible implementation of the `echo` command:

```
> clear __cmd__::echo
> extern int puts(char*) = echo;
warning: external 'echo' shadows previous undefined use of this symbol
> echo Hello, world!
Hello, world!
```

Instead of returning a result, a command function may also throw an exception. If the exception value is a string, it will be printed as an error message on standard error, using the same format as the built-in commands:

```
> error s = throw s;
> error Hello, world!
error: Hello, world!
```

You can also override a built-in command in order to provide custom functionality. In this case, the original builtin can still be executed by escaping the command name with a leading '^'. The same syntax works with the `evalcmd` function, so that a custom command can be

defined in terms of the builtin that it replaces. E.g., if we always want to invoke the `ls` command with the `-l` option, we can redefine the `ls` command as follows:

```
> ls examples/*.c
examples/poor.c  examples/sort.c
>   ls s = evalcmd $ "^ls -l "+s;
> ls examples/*.c
-rw-r--r-- 1 ag users 1883 2011-01-07 16:35 examples/poor.c
-rw-r--r-- 1 ag users 3885 2011-01-07 16:35 examples/sort.c
```

(Note that since we entered the definition of the `ls` function interactively, we need to escape the second input line above with leading whitespace, so that it's not mistaken for an invocation of the built-in `ls` command. This isn't necessary if you're using the alternative "escape" command syntax described in Command Syntax.)

To do more interesting things, you should take a look at the reflection capabilities discussed in the Macros section, which open up endless possibilities for commands to inspect and manipulate the running program in an interactive fashion. For instance, let's define a variation of the built-in `clear` command which allows us to delete a specific rule rather than an entire function definition:

```
namespace __cmd__;

clr s = case val $ "'(0 with "+s+" end)" of
  '(0 __with__ [r]) = del_fundef r;
  _ = throw "bad rule syntax";
end;
```

Note that we employ a little trick here to have `val` do all the hard work of parsing the rule specified as argument to the command, in order to translate the Pure rule syntax to the special meta representation used by `del_fundef`. The following example shows our `clr` command in action:

```
> namespace;
> fact n = 1 if n<=0;
>        = n*fact (n-1) otherwise;
> show fact
fact n = 1 if n<=0;
fact n = n*fact (n-1);
> clr fact n = 1 if n<=0;
> show fact
fact n = n*fact (n-1);
```

Here's another useful command `apropos` which quickly summarizes the information available on a given symbol (as reported by the `show` and `help_index` commands):

```
namespace __cmd__;

apropos s = case catmap descr $ split "\n" $ evalcmd $ "show -s "+s of
  [] = throw $ "undefined symbol '"+s+"'";
  info = s+" is a "+join " and a " info+". \
Type 'show "+s+"' for more information."+
```

```
(if null (evalcmd $ "help_index "+s) then "" else
"\nDocumentation for this symbol is available. Type 'help "+s+"'.");
end with
  descr info = case [x | x = split " " info; ~null x] of
    t:c:_ = [symtypes!c] if s==t when
      symtypes = {"fun"=>"function","mac"=>"macro","var"=>"variable",
                  "cst"=>"constant"};
    end;
    _ = [];
  end;
end;
```

This command can be used as follows:

```
> apropos foldl
foldl is a function. Type 'show foldl' for more information.
Documentation for this symbol is available. Type 'help foldl'.
> apropos $
$ is a macro and a function. Type 'show $' for more information.
Documentation for this symbol is available. Type 'help $'.
> let x = 11;
> apropos x
x is a variable. Type 'show x' for more information.
> apropos y
apropos: undefined symbol 'y'
```

More examples can be found in the sample.purerc file distributed with the Pure interpreter.

1.11.11 Interactive Startup

In interactive mode, the interpreter runs some additional scripts at startup, after loading the prelude and the scripts specified on the command line. This lets you tailor the interactive environment to your liking.

The interpreter first looks for a .purerc file in the user's home directory (as given by the `HOME` environment variable) and then for a .purerc file in the current working directory. These are just ordinary Pure scripts which may contain any additional definitions (including command definitions, as described in the previous section) that you need. The .purerc file in the home directory is for global definitions which should always be available when running interactively, while the .purerc file in the current directory can be used for project-specific definitions.

Finally, you can also have a .pure initialization file in the current directory, which is usually created with the `dump` command (see above). This file is loaded after the .purerc files if it is present.

The interpreter processes all these files in the same way as with the `run` command (see Interactive Commands above). When invoking the interpreter, you can specify the *`--norc`* option on the command line if you wish to skip these initializations.

1.12 Batch Compilation

The interpreter's `-c` option provides a means to turn Pure scripts into standalone executables. This feature is still a bit experimental. In particular, note that the compiled executable is essentially a *static snapshot* of your program which is executed on the "bare metal", without a hosting interpreter. Only a minimal runtime system is provided. This considerably reduces startup times, but also implies some quirks and limitations as detailed below.

First and foremost, the batch compiler always reorders the code so that all toplevel expressions and `let` bindings are evaluated *after* all functions have been defined. This is done to reduce the size of the output executable, so that there's only a *single* snapshot of each function which will be used by all toplevel expressions and global variable definitions invoking the function. Therefore you should avoid code like the following:

```
let x = foo 99;
foo x = x+1;
let y = foo 99;
```

Note that if you run this through the interpreter, `x` and `y` are bound to `foo 99` and `100`, respectively, because expressions and variable definitions are executed immediately, as the program is being processed. In contrast, if the same program is batch-compiled, *both* variables will be defined *after* the definition of `foo` and thus refer to the same value `100` instead. This will rarely be a problem in practice (the above example is really rather pathological and won't usually occur in real-world programs), but to avoid these semantic differences, you'll have to make sure that expressions are evaluated *after* all functions used in the evaluation have been defined completely. (However, the batch compiler currently doesn't check this condition and will happily generate code for programs which violate it.)

Plain toplevel expressions won't be of much use in a batch-compiled program, unless, of course, they are evaluated for their side-effects. Your program will have to include at least one of these to play the role of the "main program" in your script. In most cases these expressions are best placed after all the function and variable definitions, at the end of your program.

Also note that during a batch compilation, the compiled program is actually executed as usual, i.e., the script is also run *at compile time*. This might first seem to be a big annoyance, but it actually opens the door for some powerful programming techniques like partial evaluation. It is also a necessity because of Pure's highly dynamic nature. For instance, Pure allows you to define constants by evaluating an arbitrary expression (cf. Constant Definitions), and using `eval` a program can easily modify itself in even more unforeseeable ways. Therefore pretty much anything in your program can actually depend on previous computations performed while the program is being executed. To make this work in batch-compiled scripts, the batch compiler thus executes the script as usual. The `compiling` variable can be used to check whether the script is being batch-compiled, so you can adjust to that by selectively enabling or disabling parts of the code. For instance, you will usually want to skip execution of the "main program" during batch compilation.

Last but not least, note that some parts of Pure's metaprogramming capabilities and other compile time features are disabled in batch-compiled programs:

- The `eval` function can only be used to evaluate plain toplevel expressions. You can define local functions and variables in `with` and `when` clauses inside an expression, but you can't use `eval` to define new global variables and functions. In other words, anything which changes the executing program is "verboten". Moreover, the introspective capabilities provided by `evalcmd` and similar operations (discussed under Reflection in the Macros section) are all disabled. If you need any of these capabilities, you have to run your program with the interpreter.
- Constant and macro definitions, being compile time features, aren't available in the compiled program. If you need to use these with `eval` at run time, you have to provide them through variable and function definitions instead. Also, the compiler usually strips unused functions from the output code, so that only functions which are actually called somewhere in the static program text are available to `eval`. (The `-u` option and the `--required` pragma can be used to avoid this, see Options Affecting Code Size below.)
- Code which gets executed to compute constant values at compile time will generally *not* be executed in the compiled executable, so your program shouldn't rely on side-effects of such computations (this would be bad practice anyway). There is an exception to this rule, however, namely if a constant value contains run time data such as pointers and local functions which requires an initialization at run time, then the batch compiler will generate code for that. (The same happens if the `--noconst` option is used to force computation of constant values at run time, see Options Affecting Code Size.)

What this boils down to is that in the batch-compiled program you will have to avoid anything which requires the compile time or interactive facilities of the interpreter. These restrictions only apply at run time, of course. At compile time the program *is* being executed by the full version of the interpreter so you can use `eval` and `evalcmd` in any desired way.

For most kinds of scripts, the above restrictions aren't really that much of an obstacle, or can easily be worked around. For the few scripts which actually need the full dynamic capabilities of Pure you'll just have to run the script with the interpreter. This isn't a big deal either, only the startup will be somewhat slower because the script is compiled on the fly. Once the JIT has done its thing the "interpreted" script will run every bit as fast as the "compiled" one, since in fact *both* are compiled (only at different times) to exactly the same code!

1.12.1 Example

For the sake of a concrete example, consider the following little script:

```
using system;

fact n = if n>0 then n*fact (n-1) else 1;

main n = do puts ["Hello, world!", str (map fact (1..n))];

if argc<=1 then () else main (sscanf (argv!1) "%d");
```

When invoked from the command line, with the number `n` as the first parameter, this program will print the string `"Hello, world!"` and the list of the first `n` factorials:

```
$ pure -x hello.pure 10
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Note the condition on `argc` in the last line of the script. This prevents the program from producing an exception if no command line parameters are specified, so that the program can also be run interactively:

```
$ pure -i -q hello.pure
> main 10;
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
()
> quit
```

To turn the script into an executable, we just invoke the Pure interpreter with the `-c` option, using the `-o` option to specify the desired output file name:

```
$ pure -c hello.pure -o hello
$ ./hello 10
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Next suppose that we'd like to supply the value `n` at *compile* rather than run time. To these ends we want to turn the value passed to the `main` function into a compile time constant, which can be done as follows:

```
const n = if argc>1 then sscanf (argv!1) "%d" else 10;
```

(Note that we provide `10` as a default if `n` isn't specified on the command line.)

Moreover, in such a case we usually want to skip the execution of the main function at compile time. To these ends, the predefined `compiling` variable holds a truth value indicating whether the program is actually running under the auspices of the batch compiler, so that it can adjust accordingly. In our example, the evaluation of `main` becomes:

```
if compiling then () else main n;
```

Our program now looks as follows:

```
using system;

fact n = if n>0 then n*fact (n-1) else 1;

main n = do puts ["Hello, world!", str (map fact (1..n))];

const n = if argc>1 then sscanf (argv!1) "%d" else 10;
if compiling then () else main n;
```

This script "specializes" `n` to the first (compile time) parameter when being batch-compiled,

and it still works as before when we run it through the interpreter in both batch and interactive mode, too:

```
$ pure -i -q hello.pure
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
> main 5;
Hello, world!
[1,2,6,24,120]
()
> quit

$ pure -x hello.pure 7
Hello, world!
[1,2,6,24,120,720,5040]

$ pure -o hello -c -x hello.pure 7

$ ./hello
Hello, world!
[1,2,6,24,120,720,5040]
```

In addition, there's also a *compile time* check analogous to the `compiling` variable, which indicates whether the source script is being run normally or in a batch compilation; see Conditional Compilation. We might employ this as follows, replacing the last line of the script with this:

```
#! --if compiled
if compiling then () else main n;
#! --else
if argc>1 then main n else puts "Try 'main n' where n is a number.";
#! --endif
```

The code in the `--if compiled` section, which is the same as before, is now only executed during batch compilation and in the compiled executable. If we run the script normally, in the interpreter, the code in the `--else` section, which just prints a welcome message if no arguments are given on the command line, is executed instead. So we now actually have *four* different code paths, depending on whether the script is run normally, with or without arguments, or in a batch compilation, or as a native executable. This kind of setup is useful if the script is to be run both interactively and non-interactively in the interpreter while developing it, but once the script is finished it gets compiled and installed as a native executable.

```
$ pure -i -q hello.pure
Try 'main n' where n is a number.
> main 5;
Hello, world!
[1,2,6,24,120]
()
> quit

$ pure -x hello.pure 7
```

```
Hello, world!
[1,2,6,24,120,720,5040]

$ pure -o hello -c -x hello.pure

$ ./hello
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

You'll rarely need an elaborate setup like this, most of the time something like our simple first example will do the trick. But, as you've seen, Pure can easily do it.

1.12.2 Options Affecting Code Size

By default, the batch compiler strips unused functions from the output code, to keep the code size small. You can disable this with the `-u` option, in which case the output code includes *all* functions defined in the compiled program, the prelude and any other module imported with a `using` clause, even if they don't seem to be used anywhere. This considerably increases compilation times and makes the compiled executable much larger. For instance, on a 64 bit Linux systems with ELF binaries the executable of our hello.pure example is about thrice as large:

```
$ pure -o hello -c -x hello.pure 7 && ls -l hello
-rwxr-xr-x 1 ag users 178484 2010-01-12 06:21 hello
$ pure -o hello -c -u -x hello.pure 7 && ls -l hello
-rwxr-xr-x 1 ag users 541941 2010-01-12 06:21 hello
```

(Note that even the stripped executable is fairly large when compared to compiled C code, as it still contains the symbol table of the entire program, which is needed by the runtime environment.)

Stripped executables should be fine for most purposes, but you have to be careful when using `eval` in your compiled program. The compiler only does a *static* analysis of which functions might be reached from the initialization code (i.e., toplevel expressions and `let` bindings). It does *not* take into account code run via the `eval` routine. Thus, functions used only in `eval`ed code will be stripped from the executable, as if they were never defined at all. If such a function is then being called using `eval` at runtime, it will evaluate to a plain constructor symbol.

If this is a problem then you can either use the `-u` option to produce an unstripped executable, or you can force specific functions to be included in the stripped executable with the `--required` pragma (cf. Code Generation Options). For instance:

```
#! --required foo
foo x = bar (x-1);
eval "foo 99";
```

There is another code generation option which may have a substantial effect on code size, namely the `--noconst` option. Normally, constant values defined in a `const` definition are

precomputed at compile time and then stored in the generated executable; this reduces startup times but may increase the code size considerably if your program contains big constant values such as lists. If you prefer smaller executables then you can use the *--noconst* option to force the value of the constant to be recomputed at run time (which effectively turns the constant into a kind of read-only variable). For instance:

```
#! --noconst
const xs = 1L..100000L;
sum = foldl (+) 0;

using system;
puts $ str $ sum xs;
```

On my 64 bit Linux system this produces a 187115 bytes executable. Without *--noconst* the code becomes almost an order of magnitude larger in this case (1788699 bytes). On the other hand, the smaller executable also takes a little longer to run since it must first recompute the value of the list constant at startup. So you have to consider the tradeoffs in a given situation. Usually big executables aren't much of a problem on modern operating systems, but if your program contains a lot of big constants then this may become an important consideration. However, if a constant value takes a long time to compute then you'll be better off with the default behaviour of precomputing the value at compile time.

1.12.3 Other Output Code Formats

Note that while the batch compiler generates native executables by default, it can just as well create object files which can be linked into other C/C++ programs and libraries:

```
$ pure -o hello.o -c -x hello.pure 7
```

The .o extension tells the compiler that you want an object file. When linking the object module, you also need to supply an initialization routine which calls the `__pure_main__` function in hello.o to initialize the compiled module. This routine is declared in C/C++ code as follows:

```
extern "C" void __pure_main__(int argc, char** argv);
```

As indicated, `__pure_main__` is to be invoked with two parameters, the argument count and `NULL`-terminated argument vector which become the `argc` and the `argv` of the Pure program, respectively. (You can also just pass 0 for both arguments if you don't need to supply command line parameters.) The purpose of `__pure_main__` is to initialize a shell instance of the Pure interpreter which provides the minimal runtime support necessary to execute the Pure program, and to invoke all "initialization code" (variable definitions and toplevel expressions) of the program itself.

A minimal C `main` function which does the job of initializing the Pure module looks as follows:

```
extern void __pure_main__(int argc, char** argv);
```

```
int main(int argc, char** argv)
{
  __pure_main__(argc, argv);
  return 0;
}
```

If you link the `main` routine with the Pure module, don't forget to also pull in the Pure runtime library. Assuming that the above C code is in pure_main.c:

```
$ gcc -c pure_main.c -o pure_main.o
$ g++ -o hello hello.o pure_main.o -lpure
$ ./hello
Hello, world!
[1,2,6,24,120,720,5040]
```

(The C++ compiler is used as the linker here so that the standard C++ library gets linked in, too. This is necessary because Pure's runtime library is actually written in C++.)

In fact, this is pretty much what `pure -c` actually does for you when creating an executable.

If your script loads dynamic libraries (`using "lib:...";`) then you'll also have to link with those; *all* external references have to be resolved at compile time. This is taken care of automatically when creating executables. Otherwise it is a good idea to run `pure -c` with the `-v0100` verbosity option so that it prints the libraries to be linked (in addition to the commands which are invoked in the compilation process):

```
$ pure -v0100 -c hello.pure -o hello.o
opt -f -std-compile-opts hello.o.bc | llc -f -o hello.o.s
gcc -c hello.o.s -o hello.o
Link with: g++ hello.o -lpure
```

Well, we already knew that, so let's consider a slightly more interesting example from Pure's ODBC module:

```
$ pure -v0100 -c pure-odbc/examples/menagerie.pure -o menagerie.o
opt -f -std-compile-opts menagerie.o.bc | llc -f -o menagerie.o.s
gcc -c menagerie.o.s -o menagerie.o
Link with: g++ menagerie.o /usr/local/lib/pure/odbc.so -lpure
$ g++ -shared -o menagerie.so menagerie.o /usr/local/lib/pure/odbc.so -lpure
```

Note that the listed link options are necessary but might not be sufficient; `pure -c` just makes a best guess based on the Pure source. On most systems this will be good enough, but if it isn't, you can just add options to the linker command as needed to pull in additional required libraries.

As this last example shows, you can also create shared libraries from Pure modules. However, on some systems (most notably x86_64), this requires that you pass the *`-fPIC`* option when batch-compiling the module, so that position-independent code is generated:

```
$ pure -c -fPIC pure-odbc/examples/menagerie.pure -o menagerie.o
```

Note that even when building a shared module, you'll have to supply an initialization routine which calls `__pure_main__` somewhere.

Also note that since Pure doesn't support separate compilation in the present implementation, if you create different shared modules like this, each will contain their own copy all the required Pure functions from the prelude and other imported Pure modules. This becomes a problem when trying to link several separate batch-compiled modules into the same executable or library, because you'll get many name clashes for routines present in different modules (including the `__pure_main__` entry point). To prevent this, the batch compiler can be invoked with the *`--main`* option to explicitly set a name for the main entry point. For instance:

```
$ pure -c hello.pure -o hello.o --main __hello_main__
```

This has two effects. First, the main entry point will be called whatever you specified with *`--main`*, so you have to call this function instead of `__pure_main__` to initialize the module. Second, if *`--main`* is specified, then all Pure functions in the module will be changed to internal linkage (like `static` functions in C) to prevent any possible name clashes between different modules. (Alas, this also makes it impossible to employ `pure_funcall` to call Pure functions directly from C, as described in the following section, so you'll have to use other runtime routines such as `pure_eval` or `pure_appl` to achieve this in an indirect way.)

Last but not least, `pure -c` can also generate just plain LLVM assembler code:

```
pure -c hello.pure -o hello.ll
```

Note the .ll extension; this tells the compiler that you want an LLVM assembler file. An LLVM bitcode file can be created just as easily:

```
pure -c hello.pure -o hello.bc
```

In these cases you'll have to have to handle the rest of the compilation yourself. This gives you the opportunity, e.g., to play with special optimization and code generation options provided by the LLVM toolchain. Please refer to the LLVM documentation (in particular, the description of the opt and llc programs) for details.

1.12.4 Calling Pure Functions From C

Another point worth mentioning here is that you can't just call Pure functions in a batch-compiled module directly. That's because in order to call a Pure function, at least in the current implementation, you have to set up a Pure stack frame for the function. However, there's a convenience function called `pure_funcall` in the runtime API to handle this. This function takes a pointer to the Pure function, the argument count and the arguments themselves (as `pure_expr*` objects) as parameters. For instance, here is a pure_main.c module which can be linked against the hello.pure program from above, which calls the `fact` function from the Pure program:

```
#include <stdio.h>
#include <pure/runtime.h>
```

```
extern void __pure_main__(int argc, char** argv);
extern pure_expr *fact(pure_expr *x);

int main()
{
  int n = 10, m;
  __pure_main__(0, NULL);
  if (pure_is_int(pure_funcall(fact, 1, pure_int(n)), &m))
    printf("fact %d = %d\n", n, m);
  return 0;
}
```

And here's how you can compile, link and run this program:

```
$ pure -o hello.o -c -x hello.pure 7
$ gcc -o pure_main.o -c pure_main.c
$ g++ -o myhello hello.o pure_main.o -lpure
$ ./myhello
Hello, world!
[1,2,6,24,120,720,5040]
fact 10 = 3628800
```

Note that the first two lines are output from the Pure program; the last line is what gets printed by the `main` routine in pure_main.c.

1.13 Caveats and Notes

This section is a grab bag of casual remarks, useful tips and tricks, and information on common pitfalls, quirks and limitations of the current implementation and how to deal with them.

1.13.1 Etymology

People keep asking me what's so "pure" about Pure. The long and apologetic answer is that Pure tries to stay as close as possible to the spirit of term rewriting without sacrificing practicality. Pure's term rewriting core is in fact purely functional. It's thus possible and in fact quite easy to write purely functional programs in Pure, and you're encouraged to do so whenever this is reasonable. On the other hand, Pure doesn't get in your way if you want to call external operations with side effects; after all, it does allow you to call any C function at any point in a Pure program.

The short answer is that I simply liked the name, and there wasn't any programming language named "Pure" yet (quite a feat nowadays), so there's one now. If you insist on a (recursive) backronym, just take "Pure" to stand for the "Pure universal rewriting engine".

1.13.2 Backward Compatibility

Pure is based on the author's earlier Q language, but it offers many new and powerful features and programs run much faster than their Q equivalents. The language also went through a thorough facelift in order to modernize the syntax and make it more similar to other modern-style functional languages, in particular Miranda and Haskell. Thus porting Q scripts to Pure often involves a substantial amount of manual work, but it can (and has) been done.

Since its modest beginnings in April 2008, Pure has gone through a lot of major and minor revisions which raise various backward compatibility issues. We document these in the following, in order to facilitate the porting of older Pure scripts.

Pure 0.7 introduced built-in matrix structures, which called for some minor changes in the syntax of comprehensions and arithmetic sequences. Specifically, the template expression and generator/filter clauses of a comprehension are now separated with `|` instead of `;`. Moreover, arithmetic sequences with arbitrary stepsize are now written `x:y..z` instead of `x,y..z`, and the '`..`' operator now has a higher precedence than the '`,`' operator. This makes writing matrix slices like `x!!(i..j,k..l)` much more convenient.

In Pure 0.13 the naming of the logical and bitwise operations was changed, so that these are now called `~`, `&&`, `||` and `not`/`and`/`or`, respectively. (Previously, `~` was used for bitwise, `not` for logical negation, which was rather inconsistent, albeit compatible with the naming of the `not` operation in Haskell and ML.) Also, to stay in line with this naming scheme, inequality was renamed to `~=` (previously `!=`).

Pure 0.14 introduced the namespaces feature. Consequently, the scope of private symbols is now confined to a namespace rather than a source module; scripts making use of private symbols need to be adapted accordingly. Also note that syntax like `foo::int` may now also denote a qualified symbol rather than a tagged variable, if `foo` has been declared as a namespace. You can work around such ambiguities by renaming the variable, or by placing spaces around the '`::`' delimiter (these aren't permitted in a qualified symbol, so the construct `foo :: int` is always interpreted as a tagged variable, no matter whether `foo` is also a valid namespace).

Pure 0.26 extended the namespaces feature to add support for hierarchical namespaces. This means that name lookup works in a slightly different fashion now (see Hierarchical Namespaces for details), but old code which doesn't use the new feature should continue to work unchanged.

Pure 0.26 also changed the `nullary` keyword to `nonfix`, which is more consistent with the other kinds of fixity declarations. Moreover, the parser was enhanced so that it can cope with a theoretically unbounded number of precedence levels, and the system of standard operators in the prelude was modified so that it becomes possible to sneak in new operator symbols with ease; details can be found in the Symbol Declarations section.

Pure 0.41 added support for optimization of indirect tail calls, so that any previous restrictions on the use of tail recursion in indirect function calls and mutually recursive globals have been removed. Moreover, the logical operators `&&` and `||` are now tail-recursive in their second operand and can also be extended with user-defined equations, just like the

other builtins. Note that this implies that the values returned by `&&` and `||` aren't normalized to the values 0 and 1 any more (this isn't possible with tail call semantics). If you need this then you'll have to make sure that either the operands are already normalized, or you'll have to normalize the result yourself.

Also, as of Pure 0.41 the batch compiler produces stripped executables by default. To create unstripped executables you now have to use the *-u* option, see Options Affecting Code Size for details. The *-s* option to produce stripped executables is still provided for backward compatibility, but it won't have any effect unless you use it to override a previous *-u* option.

Pure 0.43 changed the rules for looking up symbols in user-defined namespaces. Unqualified symbols are now created in the current (rather than the global) namespace by default, see Symbol Lookup and Creation for details. The *-w* option can be used to get warnings about unqualified symbols which are resolved to a different namespace than previously. It also provides a means to check your scripts for implicit declarations which might indicate missing or mistyped function symbols.

Pure 0.45 added support for checking arbitrary pointer types in the C interface, so that you don't have to worry about passing the wrong kinds of pointers to system and library routines any more. Moreover, the interpretation of numeric pointer arguments (`int*` etc.) was changed to bring them in line with the other new numeric matrix conversions (`int**` etc.). In particular, the matrix data can now be modified in-place and type checking is more strict (`int*` requires an int matrix, etc.). Also, there's now support for `argv`-style vector arguments (`char**` and `void**`). Please see the C Types section for details.

Pure 0.47 added a bunch of new features which have been on the wishlist for the forthcoming 1.0 release:

- You can now define your own interactive commands by placing suitable function definitions in the special `__cmd__` namespace; see User-Defined Commands for details.
- The syntax used to denote inline code sections was changed from `%{...%}` to `%<...%>`. This resolves an ambiguity in the syntax (note that `%{` is legal Pure syntax; it could denote a `%` operator followed by a matrix value), and also makes it easier to properly support this construct in Emacs Pure mode.
- It is now possible to declare variadic externs, so that functions like `printf` can be called without much ado; see Variadic C Functions.
- Support for simple kinds of matrix patterns like `{x,y}`, `{x::int,y}`, `{x,y;z,t}`, `{{x,y},z}` was added.
- The meaning of quoted specials such as lambdas and local definitions was changed. Previously these would be evaluated even in the middle of a quoted expression. Now they will produce a special meta representation in terms of built-in macros, in order to support the advanced metaprogramming capabilities discussed in Built-in Macros and Special Expressions and Reflection.
- Last but not least, Pure 0.47 sports a new, more flexible type tag feature which defines type tags as unary predicates implemented using normal rewriting rules; cf. section Type Rules for details. To these ends, a new keyword `type` was added (if you used this as an ordinary identifier, you will have to rename these). Note that the old-style type

tags, which were just a syntactic shortcut for "as" patterns involving unary constructor symbols, aren't supported any more, so you'll have to fix up your old scripts accordingly. To assist with this, the Pure interpreter can be run with the `-w` option in order to identify occurrences of undefined (presumably old-style) type tags. You should either change these to the corresponding "as" pattern (i.e., `x::foo` to `x@(foo _)`), or just add a proper type definition for the tag (like `type foo (foo _);`).

Pure 0.48 moved pointer arithmetic and the regex functions into separate `pointers` and `regex` modules, so you now have to import these modules if you need this functionality. It also introduced the `--defined` pragma which lets you have "defined" functions in Pure which throw an exception if they can't be applied, e.g., because they are invoked with the wrong arguments.

Pure 0.49 introduced the conditional compilation pragmas, so that simple version and system dependencies can now be handled in a more convenient way.

Pure 0.50 introduced the declaration of interface types, which make it possible to create the definition of a type from a description of its operations. To these ends, a new keyword `interface` was added to the language.

Pure 0.55 changed the default compilers for inline C, C++ and Fortran code to clang, clang++ and gfortran (with the dragonegg plugin), respectively. This was done in order to support LLVM 3.x which does not have llvm-gcc (the previous default) any more. If you're still running an older LLVM version and would like to keep using llvm-gcc, you will have to set some environment variables; please see the *installation instructions* for details.

Pure 0.56 fixed the meaning of patterns in comprehensions so that unmatched elements are now filtered out automatically, like in Haskell. The previous behaviour of raising an exception in such cases offered no real benefits and was in fact very inconvenient in most situations.

Pure 0.56 also introduced the *non-splicing vector brackets* `{| |}` as an alternative notation for nested vector structures, as well as the namespace brackets to quickly switch namespaces inside an expression. The `true` and `false` constants are now declared as `nonfix` symbols in the prelude so that they can be used in patterns, and there's a new `bool` type to check for normalized truth values and a `bool` function to convert machine integers to normalized truth values. There's also a new `enum` module to facilitate the creation of enumerated types. Moreover, interactive commands can now optionally be escaped by prefixing them with a special character (see Interactive Usage for details).

1.13.3 Error Recovery

The parser uses a fairly simplistic panic mode error recovery which tries to catch syntax errors at the toplevel only. This seems to work reasonably well, but might catch some errors much too late. Unfortunately, Pure's terseness makes it rather difficult to design a better scheme. As a remedy, the parser accepts an empty definition (just `;` by itself) at the toplevel only. Thus, in interactive usage, if the parser seems to eat away your input without doing anything, entering an extra semicolon or two should break the spell, putting you back at the toplevel where you can start typing the definition again.

1.13.4 Splicing Tuples and Matrices

The "splicing" of tuples and matrices is probably one of Pure's most controversial features. By this we mean that tuples and matrices get flattened out when they are combined. For instance:

```
> (1,2,3),4,(5,6);
1,2,3,4,5,6
> {{1,2,3},4,{5,6}};
{1,2,3,4,5,6}
> {{a,b;c,d},{x;y}}
{a,b,x;c,d,y}
```

This kind of behaviour is also known from Perl and MATLAB/Octave. Users familiar with these languages often find it convenient, but it certainly gets in the way if you want to nest these structures. Fortunately, there are some remedies for the most common cases where you'd want to do this. Specifically, for the case of vectors the prelude defines the *non-splicing vector brackets* which make it easy to construct nested vectors; these are often used to represent multi-dimensional indexable collections in Pure. For instance:

```
> {|{1,2,3},4,{5,6}|};
{{1,2,3},4,{5,6}}
> {|{a,b;c,d},{x;y}|};
{{a,b;c,d},{x;y}}
> ans!0!(1,1);
d
```

Nothing like this is available for tuples, though, so you'll have to use lists instead if you need nestability. Note that the deeper reason behind the non-nestability of tuples is the right-recursive nature of tuples combined with the fact that there aren't any real 1-tuples in Pure (`(x)` is just `x`). This implies that you can't have a nested tuple in the last component of a tuple, no matter how hard you try to prevent the splicing, e.g., by quoting. `x,(y,z)` is always just the triple `x,y,z`.

One might consider this a defect in Pure's tuple data structure. But Pure already has a nestable kind of tuples (lists), so it would be rather pointless to have yet another isomorphic data structure with just slightly different syntax. Instead Pure gives you the choice between two kinds of list-like data structures, one which nests, and one which doesn't but has other interesting properties.

1.13.5 With and when

Another common source of confusion is that Pure provides two different constructs to bind local function and variable symbols, respectively. This distinction is necessary because Pure does not segregate defined functions and constructors, and thus there is no magic to figure out whether an equation like `foo x = y` by itself is meant as a definition of a function `foo` with formal parameter `x` and return value `y`, or a pattern binding defining the local variable `x` by matching the pattern `foo x` against the value of `y`. The `with` construct does the former, `when` the latter.

Also note that the function definitions in a `with` clause are all done simultaneously (and can thus be mutually recursive), while the individual variable definitions and expressions in a `when` clause are executed in order. This works in exactly the same fashion as `letrec` and `let` in Scheme. (As a mnemonic, consider that `when` conveys a sense of time, so its parts are "executed in sequence".)

The sequential execution aspect of `when` is rather important in Pure, because it enables you to do a series of "actions" (variable bindings and expression evaluations) in sequence by simply enclosing it in a `when` clause. This provides the Pure programmer with a useful and familiar bit of imperative "look and feel" (even though the `when` clause itself works in a purely functional way). For instance, suppose that we'd like to define a function which opens a file, checks that the file was opened successfully and throws an exception otherwise, outputs a message to indicate which file was opened, and finally returns the contents of the file as a string. The easiest way to do this in Pure is as follows:

```
using system;

read_file name::string = s when
  fp = fopen name "r";
  pointerp fp || throw (sprintf "%s: %s" (name,strerror errno));
  printf "opened file %s\n" name;
  s = fget fp;
end;
```

Another bit of syntax that may take getting used to is that `with` and `when` clauses are tacked on to the end of the expression they belong to. This mimics mathematical language and supposedly makes it easier to read and understand a definition, because you're told right up front *what* is to be computed, before going into the details of *how* the computation is performed. Unfortunately, this style differs considerably from other block-structured programming languages, which often place local definitions in front of the code they apply to. Pure doesn't offer any special syntax for this, but note that you can always write a `when` or `with` clause in the following style which places the "body" at the bottom:

```
result when
  y = foo (x+1);
  z = bar y;
  result = baz z;
end;
```

This can be read and written more or less like a `let` expression in Scheme or ML, except that the name of the result is given explicitly at the beginning. However, this style doesn't really save you either if you need several sections with both local functions and variables. In this case you'll just have to bite the bullet and arrange the `with` and `when` clauses the way that Pure wants them. That is, first come the local variables used in the right-hand side, then the local functions needed to compute those variables, then maybe another section with local variables needed by those functions, etc. When looking at such a complicated series of definitions, it sometimes helps to read the `with` and `when` blocks "in reverse", i.e., from bottom to top, which is the order in which they will actually be executed.

1.13.6 Non-Linear Patterns

As explained in section Patterns, Pure allows multiple occurrences of the same variable in a pattern (so-called non-linearities):

```
foo x x = x;
```

This rule will only be matched if both occurrences of `x` are bound to the same value. More precisely, the two instances of `x` will checked for syntactic equality during pattern matching, using the `same` primitive provided by the prelude. This may need time proportional to the sizes of both argument terms, and thus become quite costly for big terms. In fact, `same` might not even terminate at all if the compared terms are both infinite lazy data structures, such as in `foo (1..inf) (1..inf)`. So you have to be careful to avoid such uses.

When using non-linearities in conjunction with "as" patterns, you also have to make sure that the "as" variable does not occur inside the corresponding subpattern. Thus a definition like the following is illegal:

```
> foo xs@(x:xs) = x;
<stdin>, line 1: error in pattern (recursive variable 'xs')
```

The explanation is that such a pattern couldn't possibly be matched by a finite list anyway. Indeed, the only match for `xs@(x:xs)` would be an infinite list of `x`'s, and there's no way that this condition could be verified in a finite amount of time. Therefore the interpreter reports a "recursive variable" error in such situations.

1.13.7 "As" Patterns

In the current implementation, "as" patterns cannot be placed on the "spine" of a function definition. Thus rules like the following, which have the pattern somewhere in the head of the left-hand side, will all provoke an error message from the compiler:

```
a@foo x y   = a,x,y;
a@(foo x) y = a,x,y;
a@(foo x y) = a,x,y;
```

This is because the spine of a function application is not available when the function is called at runtime. "As" patterns in pattern bindings (`let`, `const`, `case`, `when`) are not affected by this restriction since the entire value to be matched is available at runtime. For instance:

```
> case bar 99 of y@(bar x) = y,x+1; end;
bar 99,100
```

1.13.8 "Head = Function" Pitfalls

The "head = function" rule stipulates that the head symbol `f` of an application `f x1 ... xn` occurring on (or inside) the left-hand side of an equation, variable binding, or pattern-matching lambda expression, is always interpreted as a literal function symbol (not a vari-

able). This implies that you cannot match the "function" component of an application against a variable, at least not directly. An anonymous "as" pattern like `f@_` does the trick, however, since the anonymous variable is always recognized, even if it occurs as the head symbol of a function application. Here's a little example which demonstrates how you can convert a function application to a list containing the function and all arguments:

```
> foo x = a [] x with a xs (x@_ y) = a (y:xs) x; a xs x = x:xs end;
> foo (a b c d);
[a,b,c,d]
```

This may seem a little awkward, but as a matter of fact the "head = function" rule is quite convenient, since it covers the common cases without forcing the programmer to declare variable or constructor symbols (other than nonfix symbols). On the other hand, generic rules operating on arbitrary function applications are not all that common, so having to "escape" a variable using the anonymous "as" pattern trick is a small price to pay for that convenience.

Sometimes you may also run into the complementary problem, i.e., to match a function argument against a given function. Consider this code fragment:

```
foo x = x+1;
foop f = case f of foo = 1; _ = 0 end;
```

You might expect `foop` to return true for `foo`, and false on all other values, but in reality `foop` will always return true! In fact, the Pure compiler will warn you about the second rule of the `case` expression not being used at all:

```
> foop 99;
warning: rule never reduced: _ = 0;
1
```

This is again due to the "head = function" rule; `foo` is neither the head symbol of a function application nor a `nonfix` symbol here, so it is considered a variable, even though it is defined as a global function elsewhere. (As a matter of fact, this is rather useful, since otherwise a rule like `f g = g+1` would suddenly change meaning if you happen to add a definition like `g x = x-1` somewhere else in your program, which certainly isn't desirable.)

A possible workaround is to "escape" the function symbol using an empty namespace qualifier:

```
foop f = case f of ::foo = 1; _ = 0 end;
```

This trick works in `case` expressions and function definitions, but fails in circumstances in which qualified variable symbols are permitted (i.e., in variable and constant definitions). A better solution is to employ the syntactic equality operator `===` defined in the prelude to match the target value against the function symbol. This allows you to define the `foop` predicate as follows:

```
> foop f = f===foo;
> foop foo, foop 99;
1,0
```

Another way to deal with the situation would be to just declare `foo` as a nonfix symbol. However, this makes the `foo` symbol "precious", i.e., after such a declaration it cannot be used as a local variable anymore. It's usually a good idea to avoid that kind of thing, at least for generic symbols, so the above solution is preferred in this case.

1.13.9 Defined Functions

As explained in Definitions and Expression Evaluation, Pure doesn't really distinguish "constructors" from "defined functions" and thus allows any function symbol to become part of a normal form expression yielded by an evaluation. This behaviour follows the usual semantics of (typeless) term rewriting and is actually quite useful if you also want to evaluate expressions symbolically.

However, this becomes a nuisance if you really expect the given function to reduce to something else, and just accidentally supplied the wrong arguments to the function. Especially annoying in this respect are functions involving side effects:

```
> using system;
> puts 99;
puts 99
```

Here we accidentally specified a number (rather than a string) as the argument of the `puts` function. This kind of error can easily be spotted if the function is invoked interactively, but it may well go unnoticed if the call is buried deeply in a big program which runs unattended (in batch mode).

As a remedy, Pure 0.48 introduces the *`--defined`* pragma (cf. Code Generation Options) which allows you to explicitly declare a function symbol as a "defined" function, so that it will raise a proper exception when the defining equations (or, as it were, the external definition) of the function are not applicable to the subject expression:

```
> #! --defined puts
> puts 99;
<stdin>, line 4: unhandled exception 'failed_match' while evaluating 'puts 99'
```

This is the same kind of `failed_match` exception that you'll get, e.g., if the subject term fails to match all patterns in a `case` construct, cf. Exception Handling. However, note that the exception will only be generated if the symbol actually has any defining equations, so a "pure constructor" (i.e., a symbol without defining equations) will still return a normal form even if it is also declared *`--defined`*:

```
> #! --defined foo
> foo bar;
foo bar
```

Nevertheless, the *`--defined`* pragma will be recorded and take effect as soon as you add an equation for the function:

```
> foo x::int = x+1;
> foo bar;
<stdin>, line 4: unhandled exception 'failed_match' while evaluating 'foo bar'
```

There's also a *--nodefined* pragma which reverts the function to the default behaviour of returning normal forms:

```
> #! --nodefined foo
> foo bar;
foo bar
```

As indicated, the *--defined* and *--nodefined* pragmas can be invoked freely at any time, and the interpreter takes care that the affected function is recompiled automatically as needed.

Please note that the *--defined* pragma is still considered experimental. It interferes with Pure's symbolic evaluation capabilities, so the pragma isn't currently used in the standard library and we recommend that programmers shouldn't use it in a careless fashion either. However, while most error conditions stemming from unexpected normal forms can also be caught with diligent unit testing, the pragma can sometimes save you some time and trouble, especially when testing programs which are to be executed mostly in batch mode. Future versions of the interpreter might also make good use of this pragma for static checks and optimization purposes.

1.13.10 Stack Size and Tail Recursion

Pure programs may need a considerable amount of stack space to handle recursive function and macro calls, and the interpreter itself also takes its toll. So you should configure your system accordingly (8 MB of stack space is recommended for 32 bit systems, systems with 64 bit pointers probably need more). If the `PURE_STACK` environment variable is defined, the interpreter performs advisory stack checks on function entry and raises a Pure exception if the current stack size exceeds the given limit. The value of `PURE_STACK` should be the maximum stack size in kilobytes. Please note that this is only an advisory limit which does not change the program's physical stack size. Your operating system should supply you with a command such as ulimit(1) to set the real process stack size. (The `PURE_STACK` limit should be a little less than that, to account for temporary stack usage by the interpreter itself.)

Like Scheme, Pure does proper tail calls (if LLVM provides that feature on the platform at hand), so tail-recursive definitions should work fine in limited stack space. For instance, the following little program will loop forever if your platform supports the required optimizations:

```
loop with loop = loop end;
```

This also works if your definition involves function parameters, guards and multiple equations, of course. Moreover, conditional expressions (`if-then-else`) are tail-recursive in both branches, and the logical operators `&&` and `||`, as well as the sequence operator `$$`, are tail-recursive in their second operand.

In addition, the Pure compiler also does a specialized form of tail recursion optimization for type definition rules. Due to the special way in which type tags are processed, however, the amount of optimization performed in this case is somewhat limited; see Recursive Types below.

Finally, note that tail call optimization is *always* disabled if the debugger is enabled (-g). This makes it much easier to debug programs, but means that you may run into stack overflows when debugging a program that does deep tail recursion.

1.13.11 Handling of Asynchronous Signals

As described in section Exception Handling, signals delivered to the process can be caught and handled with Pure's exception handling facilities. This has its limitations, however. Since Pure code cannot be executed directly from a C signal handler, checks for pending signals are only done on function entry. This means that in certain situations (such as the execution of an external C routine), delivery of a signal may be delayed by an arbitrary amount of time. Moreover, if more than one signal arrives between two successive signal checks, only the last one will be reported in the current implementation.

When delivering a signal which has been remapped to a Pure exception, the corresponding exception handler (if any) will be invoked as usual. Further signals are blocked while the exception handler is being executed.

A fairly typical case is that you have to handle signals in a tail-recursive function. This can be done with code like the following:

```
using system;

// Remap some common POSIX signals.
do (trap SIG_TRAP) [SIGHUP, SIGINT, SIGTERM];

loop = catch handler process $$ loop
with handler (signal k) = printf "Hey, I got signal %d.\n" k end;
process = sleep 1; // do something
```

Running the above `loop` function enters an endless loop reporting all signals delivered to the process. Note that to make this work, the tail-recursive invocation of `loop` must immediately follow the signal-handling code, so that signals don't escape the exception handler.

Of course, in a real application you'd probably want the `loop` function to carry around some data to be processed by the `process` routine, which then returns an updated value for the next iteration. This can be implemented as follows:

```
loop x = loop (catch handler (process x))
with handler (signal k) = printf "Hey, I got signal %d.\n" k $$ 0 end;
process x = printf "counting: %d\n" x $$ sleep 1 $$ x+1;
```

1.13.12 Recursive Types

Using the facilities described in Type Rules, type tags can easily be defined in a recursive fashion. In simple cases, the compiler can optimize such definitions so that they are executed in constant stack space, just like ordinary tail-recursive functions. The main difference here is that the recursion already takes place during *matching*, i.e., on the *left-hand* side of a rule, since this is where type predicates are normally invoked. This also limits the amount of tail recursion optimization available on type rules, as detailed below.

For instance, the following `rlist` type from the prelude is defined in such a way that it only matches "proper" lists which have list values in all their tails (and are thus terminated by the empty list).

```
type rlist [] | rlist (x : xs::rlist);
```

Note that this type definition recurses in the *last* `rlist` tag of the *last* rule of the type. If tail calls are supported by the host implementation (cf. Stack Size and Tail Recursion), the compiler makes sure that such definitions are safe to use even if the recursion may go arbitrarily deep. For instance:

```
> typep rlist (1..10000000);
1
```

The precise rules for tail-recursive type definitions are as follows:

- The *last* rule of the type must have a trivial right-hand side (either just `true` or missing) and must be *directly* recursive in the *last* type tag on the left-hand side of the rule.
- The rule may not contain any non-linearities. (That's because these are always checked *after* the type guards for efficiency.)

While these are rather strict requirements, they work reasonably well for simple recursive types such as the recursive list type above. More general recursion in types will not be optimized by the compiler, however, and may thus be subject to stack overflows. For instance, consider the following binary tree type:

```
nonfix nil;
type tree nil | tree (bin x l::tree r::tree);
```

This is a perfectly legal type definition, and the recursion in the last `tree` tag of the second rule will indeed be optimized away. However, the second rule also recurses on the *first* `tree` tag which will cause trouble if there are long chains of left branches in a tree. For instance:

```
> mktree xs = foldr (\x t->bin x t nil) nil xs;
> mktree [];
nil
> mktree [1,2,3];
bin 1 (bin 2 (bin 3 nil nil) nil) nil
> typep tree (mktree []);
1
> typep tree (mktree [1,2,3]);
1
```

```
> typep tree (mktree (1..10000));
<stdin>, line 6: unhandled exception 'stack_fault' while evaluating
'typep tree (mktree (1..10000))'
```

To avoid deep recursion in such cases it is necessary to implement the type using a general predicate, which handles the recursion by transforming it into a tail-recursive form using a technique like continuation passing.

There's yet another important issue with recursive type definitions, namely the *time* it takes to check the definition. In the above example, checking `rlist` takes O(n) time, where n is the size of the list. This will have dire consequences if you do this check repeatedly while traversing a list, as in the following `sum` function:

```
sum xs::rlist = if null xs then 0 else head xs+sum (tail xs);
```

As this function repeatedly checks its entire argument, the total time it takes to compute the sum of a list this way becomes O(n^2). To see how slow this function is, just try it on successively larger lists `1..1000`, `1..2000`, etc. One way to work around this is to write a "wrapper" function which simply checks the type of its argument in advance and then invokes a "worker" function to do the actual computation:

```
sum xs::rlist = sum xs with
  sum xs = if null xs then 0 else head xs+sum (tail xs);
end;
```

This "wrapper-worker" design is quite common and useful in many situations, but it is a bit cumbersome in this specific case. An easier way is to just do the type checking in a piecemeal fashion, as the list is being traversed. To these ends, the prelude also provides a basic `list` type which is defined as follows:

```
type list [] | list (x:xs);
```

Note that the recursion is missing here and thus this type can always be checked in O(1) time, performing just a single pattern match, which is efficient. Hence, if we replace `rlist` with the `list` type in our original definition then `sum` will now run in O(n) time, as desired. On the other hand, this approach also has its drawbacks. For instance, consider:

```
> sum xs::list = if null xs then 0 else head xs+sum (tail xs);
> sum (1:2:3);
1+(2+sum 3)
```

In contrast, our wrapper-worker definition of `sum` from above returns a somewhat prettier normal form instead:

```
> clear sum
> sum xs::rlist = sum xs with
>   sum xs = if null xs then 0 else head xs+sum (tail xs);
> end;
> sum (1:2:3);
sum (1:2:3)
```

Thus the wrapper-worker approach also has its merits, and whether to use one or the other depends on the situation. Similar techniques and tradeoffs also apply to other recursive types such as trees.

1.13.13 Interfaces

Pure's implementation of interface types has some notable differences to interfaces in a statically typed language like Go. These are mostly due to Pure's dynamically typed nature.

- Nothing is known about the *return type* of an interface operation, but this is no real impediment since Pure types are all about restricting the kind of *arguments* which can be passed to a function, not their result types, so return types are irrelevant to Pure's interface types anyway.
- Pure interfaces aren't based on the notion of "methods" and therefore don't provide any kind of "method dispatch". Interface operations are just ordinary Pure functions which rely on Pure's usual pattern-matching mechanism to do the dynamic dispatch.
- Membership in interface types is decided by considering the left-hand sides of the definitions of the interface functions only. Guards are not taken into account, and thus there's no real guarantee that a member of an interface type will always be valid input to an interface function.
- Interface types work best if all interface operations are completely defined on the target data domain. This may sometimes force you to add default or error rules raising exceptions, as shown in the Interface Types section, which may interfere with symbolic evaluation (cf. Exception Handling and Defined Functions). If this is not desirable, you can also just include the missing members manually. To these ends, Pure allows an interface type to be augmented with ordinary type rules as described in Type Rules. For instance, we might also have implemented the stack type discussed in the Interface Types section as follows:

```
interface stack with
  push s::stack x;
  pop s::stack;
  top s::stack;
end;

type stack [];

push xs@[] x | push xs@(_:_) x = x:xs;
pop (x:xs) = xs;
top (x:xs) = x;
```

Pure's interface types are really a compromise between theoretical soundness and practicality. From the theoretical point of view, we'd like an interface type to be the *intersection* of the interface types for the individual interface functions. Unfortunately, the pattern set for such an intersection type might well be exponential in size. Hence the approach taken in Pure is to eliminate those candidate patterns which aren't supported by all interface functions. This can be done much more efficiently, but will in general only produce a subtype of the

intersection type. (On the other hand, this method also has the advantage that the compiler can warn about potentially missing rules in some of the interface operations. We've seen in the Interface Types section that this can be fairly useful at times.)

Another issue arises with interface operations which allow the interface type in multiple arguments. A typical example are operators:

```
interface addable with x::addable + y::addable; end;
```

In the present implementation, the pattern set will be the *union* of the pattern sets for each argument, so the above definition is in fact equivalent to:

```
interface addable with x::addable + y; x + y::addable; end;
```

This makes sense in many situations, but of course this depends on the particular operation. In some cases, you might have to decide on which argument you want to place the interface type tag, or even have different types for each possible argument position.

1.13.14 Numeric Calculations

If possible, you should decorate numeric variables on the left-hand sides of function definitions with the appropriate type tags, like `int` or `double`. This often helps the compiler to generate better code and makes your programs run faster. The `|` syntax makes it easy to add the necessary specializations of existing rules to your program. E.g., taking the polymorphic implementation of the factorial as an example, you only have to add a left-hand side with the appropriate type tag to make that definition go as fast as possible for the special case of machine integers:

```
fact n::int    |
fact n         = n*fact(n-1) if n>0;
               = 1 otherwise;
```

(This obviously becomes unwieldy if you have to deal with several numeric arguments of different types, however, so in this case it is usually better to just use a polymorphic rule.)

Also note that int (the machine integers), bigint (the GMP "big" integers) and double (floating point numbers) are all different kinds of objects. While they can be used in mixed operations (such as multiplying an int with a bigint which produces a bigint, or a bigint with a double which produces a double), the `int` tag will only ever match a machine int, *not* a bigint or a double. Likewise, `bigint` only matches bigints (never int or double values), and `double` only doubles. Thus, if you want to define a function operating on different kinds of numbers, you'll also have to provide equations for all the types that you need (or a polymorphic rule which catches them all). This also applies to equations matching against constant values of these types. In particular, a small integer constant like `0` only matches machine integers, not bigints; for the latter you'll have to use the "big L" notation `0L`. Similarly, the constant `0.0` only matches doubles, but not ints or bigints.

1.13.15 Constant Definitions

Constants differ from variables in that they cannot be redefined (that's their main purpose after all) so that their values, once defined, can be substituted into other definitions which use them. For instance:

```
> const c = 2;
> foo x = c*x;
> show foo
foo x = 2*x;
> foo 99;
198
```

While a variable can be rebound to a new value at any time, you will get an error message if you try to do this with a constant:

```
> const c = 3;
<stdin>, line 5: symbol 'c' is already defined as a constant
```

Note that in interactive mode you can work around this by purging the old definition with the `clear` command. However, this won't affect any earlier uses of the symbol:

```
> clear c
> const c = 3;
> bar x = c*x;
> show foo bar
bar x = 3*x;
foo x = 2*x;
```

(You'll also have to purge any existing definition of a variable if you want to redefine it as a constant, or vice versa, since Pure won't let you redefine an existing constant or variable as a different kind of symbol. The same also holds if a symbol is currently defined as a function or a macro.)

Constants can also be used in patterns (i.e., on the left-hand side of a rule in a definition or a `case` expression), but only if they're also declared as `nonfix`. The prelude already does this for the truth values `true` and `false` (which are in fact just `1` and `0`), so that you can write, e.g.:

```
> check false = "no"; check true = "yes";
> show check
check 0 = "no";
check 1 = "yes";
> check (5>0);
"yes"
```

Note that if `true` and `false` weren't `nonfix`, the above definition of `check` wouldn't work as intended, because the `true` and `false` symbols on the left-hand side of the two equations would be interpreted as local variables. Also note that `true` and `false` are really an exceptional case; they aren't likely to be used as variables, so the prelude can make them `nonfix` by default. In most cases the standard library refrains from declaring constant symbols as `nonfix`, so that they don't accidentally clobber variables in user code. This is the case, in

particular, for constants in the `math` module such as `e`, `pi` and `i` which are much more likely to be used as variable symbols.

As the value of a constant is known at compile time, the compiler can apply various optimizations to uses of such values. In particular, the Pure compiler inlines constant scalars (numbers, strings and pointers) by literally substituting their values into the output code. It also precomputes simple constant expressions involving only (machine) integer and double values. (The latter is called **constant folding** and can also be disabled, see the description of the *`--fold`* and *`--nofold`* pragmas for details.) Example:

```
> extern double atan(double);
> const pi = 4*atan 1.0;
> show pi
const pi = 3.14159265358979;
> foo x = 2*pi*x;
> show foo
foo x = 6.28318530717959*x;
```

Constant folding also works with conditional expressions. E.g., consider:

```
const win = index sysinfo "mingw32" >= 0;
check boy = if win then bad boy else good boy;
```

On a Linux system, this gives:

```
> show check
check boy = good boy;
```

By these means, you can employ a constant to configure your code for different environments, without any runtime penalties. Note that this only works with conditional expressions, not with guarded equations. However, in the latter case the LLVM backend still eliminates dead code automatically, so the `check` function from above could also be defined as follows:

```
check boy = bad boy if win;
          = good boy otherwise;
```

In this case the code for one of the branches of `check` will be completely eliminated, depending on the outcome of the configuration check. (The interpreter will still print both equations if you type `show check`, but only one of the branches will actually be present in the assembler code of the function; you can verify this with `show -d check`.)

For efficiency, constant aggregates (lists, tuples, matrices and other kinds of non-scalar terms) receive special treatment. Here, the constant is computed once and stored in a read-only variable which then gets looked up at runtime, just like an ordinary global variable. However, there's an important difference: If a script is batch-compiled (cf. Batch Compilation), the constant value is normally computed *at compile time only*; when running the compiled executable, the constant value is simply reconstructed, which is often much more efficient than recomputing its value. For instance, you might use this to precompute a large table whose computation may be costly or involve functions with side effects:

```
const table = [foo x | x = 1..1000000];
process table;
```

Note that this only works with `const` values which are completely determined at compile time. If a constant contains run time objects such as (non-null) pointers and (local) functions, this is impossible, and the batch compiler will instead create code to recompute the value of the constant at run time. For instance, consider:

```
const p = malloc 100;
foo p;
```

Here, the value of the pointer `p` of course critically depends on its computation (involving a side effect which sets aside a corresponding chunk of memory). It would become unusable without actually executing the initialization, so the compiler generates the appropriate run time initialization code in this case. For all practical purposes, this turns the constant into a read-only variable. (There's also a code generation option to force this behaviour even for "normal" constants for which it's not strictly necessary, in order to create smaller executables; see Options Affecting Code Size for details.)

1.13.16 External C Functions

The interpreter always takes your `extern` declarations of C routines at face value. It will not go and read any C header files to determine whether you actually declared the function correctly! So you have to be careful to give the proper declarations, otherwise your program might well give a segfault when calling the function. This problem can to some extent be alleviated by using the bitcode interface, see Importing LLVM Bitcode and Inline Code in the C Interface section. However, you always have to be careful when calling variadic C functions, as the compiler has no way of checking which combinations of extra parameters a function like `printf` is to be invoked with. (As a remedy, the standard library provides safe implementations of `printf` and other commonly used variadic functions from the C library, see the *Pure Library Manual* for details.)

Another limitation of the C interface is that it does not offer any special support for C structs and C function parameters. However, an optional addon module is available which interfaces to the libffi library to provide that kind of functionality, please see *pure-ffi* for details.

Last but not least, to make it easier to create Pure interfaces to large C libraries, there's a separate pure-gen program available at the Pure website. This program takes a C header (.h) file and creates a corresponding Pure module with definitions and `extern` declarations for the constants and functions declared in the header. Please refer to *pure-gen: Pure interface generator* for details.

1.13.17 Calling Special Forms

Special forms are recognized at compile time only. Thus the `catch` function, as well as `quote` and the operators `&&`, `||`, `$$` and `&`, are only treated as special forms in direct (saturated) calls.

They can still be used if you pass them around as function values or in partial applications, but in this case they lose all their special call-by-name argument processing.

1.13.18 Laziness

Pure does lazy evaluation in the same way as Alice ML, providing an explicit operation (&) to defer evaluation and create a "future" which is called by need. However, note that like any language with a basically eager evaluation strategy, Pure cannot really support lazy evaluation in a fully automatic way. That is, coding an operation so that it works with infinite data structures usually requires additional thought, and sometimes special code will be needed to recognize futures in the input and handle them accordingly. This can be hard, but of course in the case of the prelude operations this work has already been done for you, so as long as you stick to these, you'll never have to think about these issues. (It should be noted here that lazy evaluation has its pitfalls even in fully lazy FPLs, such as hidden memory leaks and other kinds of subtle inefficiencies or non-termination issues resulting from definitions being too lazy or not lazy enough. You can read about that in any good textbook on Haskell.)

The prelude goes to great lengths to implement all standard list operations in a way that properly deals with streams (a.k.a. lazy lists). What this all boils down to is that all list operations which can reasonably be expected to operate in a lazy way on streams, will do so. (Exceptions are inherently eager operations such as `#`, `reverse` and `foldl`.) Only those portions of an input stream will be traversed which are strictly required to produce the result. For most purposes, this works just like in fully lazy FPLs such as Haskell. However, there are some notable differences:

- Since Pure uses dynamic typing, some of the list functions may have to peek ahead one element in input streams to check their arguments for validity, meaning that these functions will be slightly more eager than their Haskell counterparts.
- Pure's list functions never produce truly cyclic list structures such as the ones you get, e.g., with Haskell's `cycle` operation. (This is actually a good thing, because the current implementation of the interpreter cannot garbage-collect cyclic expression data; please see the corresponding remarks in *Expression References* for details.) Cyclic streams such as `cycle [1]` or `fix (1:)` will of course work as expected, but, depending on the algorithm, memory usage may increase linearly as they are traversed.
- Pattern matching is always refutable (and therefore eager) in Pure. If you need something like Haskell's irrefutable matches, you'll have to code them explicitly using futures. See the definition of the `unzip` function in the prelude for an example showing how to do this.

There are two other pitfalls with lazy data structures that you should be aware of:

- Laziness and side effects don't go well together, as most of the time you can't be sure when a given thunk will be executed. So as a general guideline you should avoid side effects in thunked data structures. If you can't avoid them, then at least make sure that all accesses to the affected resources are done through a single instance of

the thunked data structure. E.g., the following definition lets you create a stream of random numbers:

```
> using math;
> let xs = [random | _ = 1..inf];
```

This works as expected if only a single stream created with `random` exists in your program. However, as the `random` function in the `math` module modifies an internal data structure to produce a sequence of pseudorandom numbers, using two or more such streams in your program will in fact modify the same underlying data structure and thus produce two disjoint subsequences of the same underlying pseudorandom sequence which might not be distributed uniformly any more.

- You should avoid keeping references to potentially big (or even infinite) thunked data structures when traversing them (unless you specifically need to memoize the entire data structure). In particular, if you assign such a data structure to a local variable, the traversal of the data structure should then be invoked as a tail call. If you fail to do this, it forces the entire memoized part of the data structure to stay in main memory while it is being traversed, leading to rather nasty memory leaks. Please see the `all_primes` function in Lazy Evaluation and Streams for an example.

1.13.19 Name Capture

As explained in the Macro Hygiene section, Pure macros are lexically scoped and thus "hygienic". So in principle Pure macros are not susceptible to name capture. However, this principle only applies to "real" block constructs, not their quoted "placeholder" representations described in Built-in Macros and Special Expressions. One (rather obscure) case which deserves special attention is the case of macros involving free variables which are being called inside quoted block constructs. Note that this corresponds to the "mild" first form of name capture described in the Macro Hygiene section. For instance, consider the following example:

```
> def G x = x+y;
> '(G 10 when y = 99 end);
G 10 __when__ [y-->99]
> eval ans;
109
```

Here the free `y` variable of the macro `G` got captured by the quoted `when` clause when the quoted expression is evaluated. This happens because, using call by value, the call `G 10` gets evaluated before the `__when__` macro. So the behaviour of the macro evaluator in this case is in fact correct; the only remedy here is to avoid macros involving free variables inside a quoted block construct. The same applies to "*`quoteargs`*" macros which quote their arguments automatically, as described in Built-in Macros and Special Expressions. On the other hand, the described behaviour might even be useful at times, to forcibly rebind a free macro variable. The following little helper macro illustrates this trick:

```
> #! --quoteargs invoke
> def invoke x = x;
> foo = invoke (G 10 when y = 99 end);
> show foo
foo = 10+y when y = 99 end;
> foo;
109
```

Besides the above form of real name capture in quoted specials, there's also a case of apparent name capture in the expression printer which isn't actually real name capture, but just looks like it was. The reason for this is that the expression printer currently doesn't check for different bindings of the same variable identifier when it prints a (compile time) expression. For instance, consider:

```
> def F x = x+y when y = x+1 end;
> foo y = F y;
> show foo
foo y = y+y when y = y+1 end;
```

This *looks* as if `y` got captured, but in fact it's not, it's just the `show` command which displays the definition in an incorrect way. You can add the `-e` option to `show` which prints the deBruijn indices of locally bound symbols, then you see that the actual bindings are all right anyway:

```
> show -e foo
foo y/*0:1*/ = y/*1:1*/+y/*0:*/ when y/*0:*/ = y/*0:1*/+1 end;
```

Note that the number before the colon is the actual deBruijn index, the sequence of bits behind it is the subterm path. Thus the first instance of `y` in `y+y` (which has a deBruijn index of 1, indicating "one environment up") actually refers to the `y` in the left-hand side `foo y`, while the second instance refers to the local binding `y = y+1` in the `when` clause.

Alas, this means that if you use `dump` to write such a definition to a text file and read it back with `run` later, then the apparent name capture becomes a real one and you'll get the wrong definition. This is an outright bug in the expression printer which will hopefully be fixed some time. But for the time being you will have to correct such glitches manually.

1.14 Author

Albert Gräf <Dr.Graef@t-online.de>, Dept. of Computer Music, Johannes Gutenberg University of Mainz, Germany.

1.15 Acknowledgements

Pure wouldn't be what it is without its users and other people interested in the language. In particular, I'd like to thank Scott E. Dillard, Rooslan S. Khayrov, Jim Pryor, Eddie Rucker,

Libor Spacek, Jiri Spitz, Peter Summerland and Sergei Winitzki for their significant contributions of code, patches and documentation. Thanks are also due to Björn Lindig, Michel Salim, Ryan Schmidt and Zhihao Yuan who maintain the Arch Linux, Fedora, OSX and FreeBSD packages and ports, as well as to Vili Aapro, Jason E. Aten, Alvaro Castro Castilla, John Cowan, Chris Double, Tim Haynes, Wm Leler, John Lunney, Roman Neuhauser and Max Wolf for suggesting improvements and pointing out shortcomings, misfeatures and outright bugs. If it wasn't for all these people and others who contribute to the lively discussions on the mailing list, this project probably wouldn't have got anywhere.

Last but not least, a big thank you goes to Chris Lattner and the entire LLVM team. LLVM really changed the game for us compiler writers, as we can now stop worrying about all the nitty-gritty details of code generation and concentrate on the design and implementation of the programming language at hand.

1.16 Copying

Pure comes with a fairly liberal license which lets you distribute your own Pure programs and extensions under a license of your choice and permits linking of commercial applications against the Pure runtime and the Pure standard library without requiring special permission. Moreover, the Pure interpreter (the `pure` main program), the Pure runtime library (`libpure`) and the Pure standard library (the Pure scripts in the `lib` folder distributed with the software) are distributed as free software, and you are welcome to modify and redistribute them under the appropriate license terms, as detailed below.

(The above explanations are not legal advice. Please read the full text of the licenses and consult qualified professional counsel for an interpretation of the license terms as they apply to you.)

The *Pure interpreter* is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

The *Pure runtime library* and the *Pure standard library* are also free software: you can redistribute them and/or modify them under the terms of the GNU *Lesser* General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Pure is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Please see the GNU General Public License and the GNU Lesser General Public License for the precise license terms. You can also find the license conditions in the COPYING and COPYING.LESSER files accompanying the software. Also, please see the source code for the copyright and license notes pertaining to individual source files which are part of this software.

Pure uses LLVM as its compiler backend. LLVM is under Copyright (c) 2003-2012 by the University of Illinois at Urbana-Champaign, and is licensed under a 3-clause BSD-style license,

please read COPYING.LLVM included in the distribution for the exact licensing terms. You can also find the LLVM license at the LLVM website.

1.17 References and Links

Aardappel Wouter van Oortmerssen's functional programming language based on term rewriting, http://wouter.fov120.com/aardappel.

Alice ML A version of ML (see below) from which Pure borrows its model of lazy evaluation, http://www.ps.uni-sb.de/alice.

Franz Baader and Tobias Nipkow *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

Bertrand Wm Leler's constraint programming language based on term rewriting, http://groups.google.com/group/bertrand-constraint. See Wm Leler: *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, 1988.

Clang The new C/C++/Objective C compiler designed specifically to work on top of LLVM, http://clang.llvm.org. Clang provides a comparatively light-weight alternative to gcc which is faster and has better and more friendly diagnostics.

DragonEgg An LLVM backend for gcc 4.5 and later, http://dragonegg.llvm.org. In contrast to llvm-gcc, DragonEgg is implemented as a plugin which hooks into your system compiler.

Faust Grame's functional DSP programming language, http://faust.grame.fr.

GNU Multiprecision Library Free library for arbitrary precision arithmetic, http://gmplib.org.

GNU Octave A popular high-level language for numeric applications and free MATLAB replacement, http://www.gnu.org/software/octave.

GNU Scientific Library A free software library for numeric applications, can be used with Pure's numeric matrices, http://www.gnu.org/software/gsl.

Go Google's Go programming language, http://golang.org.

Haskell A popular non-strict FPL, http://www.haskell.org.

LLVM The LLVM code generator framework, http://llvm.org.

LLVM-GCC An LLVM-capable compiler based on gcc, see http://llvm.org. This is based on a fairly old gcc version (4.2) and has been replaced by the DragonEgg plugin in the LLVM 3.x series.

Miranda David Turner's non-strict FPL, http://miranda.org.uk. Miranda was fairly successful in its time and one of the forerunners of Haskell.

ML A popular strict FPL. See Robin Milner, Mads Tofte, Robert Harper, D. MacQueen: *The Definition of Standard ML (Revised).* MIT Press, 1997.

Michael O'Donnell *Equational Logic as a Programming Language.* Series in the Foundations of Computing. MIT Press, Cambridge, Mass., 1985.

Q Another term rewriting language by yours truly, http://q-lang.sf.net.

Chapter 2

Pure Library Manual

Version 0.56, September 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

Copyright (c) 2009-2012 by Albert Gräf. This document is available under the GNU Free Documentation License.

This manual describes the operations in the standard Pure library, including the prelude and the other library modules which come bundled with the interpreter.

There is a companion to this manual, *The Pure Manual* which describes the Pure language and the operation of the Pure interpreter.

2.1 Prelude

The prelude defines the basic operations of the Pure language. This includes the basic arithmetic and logical operations, string, list and matrix functions, as well as the support operations required to implement list and matrix comprehensions. The string, matrix and record operations are in separate modules strings.pure, matrices.pure and records.pure, the primitive arithmetic and logical operations can be found in primitives.pure. Note that since the prelude module gets imported automatically (unless the interpreter is invoked with the `--no-prelude` option), all operations discussed in this section are normally available in Pure programs without requiring any explicit import declarations, unless explicitly noted otherwise.

2.1.1 Constants and Operators

The prelude also declares a signature of commonly used constant and operator symbols. This includes the truth values `true` and `false`.

constant `true` = 1

constant `false` = 0

These are actually just integers in Pure, but sometimes it's convenient to refer to them using these symbolic constants.

In addition, the following special exception symbols are provided:

constructor `failed_cond`
constructor `failed_match`
constructor `stack_fault`
constructor `malloc_error`

These are the built-in exception values. `failed_cond` denotes a failed conditional in guard or if-then-else; `failed_match` signals a failed pattern match in lambda, `case` expression, etc.; `stack_fault` means not enough stack space (`PURE_STACK` limit exceeded); and `malloc_error` indicates a memory allocation error.

constructor `bad_list_value` x
constructor `bad_tuple_value` x
constructor `bad_string_value` x
constructor `bad_matrix_value` x

These denote value mismatches a.k.a. dynamic typing errors. They are thrown by some operations when they fail to find an expected value of the corresponding type.

constructor `out_of_bounds`

This exception is thrown by the index operator `!` if a list, tuple or matrix index is out of bounds.

Here's the list of predefined operator symbols. Note that the parser will automagically give unary minus the same precedence level as the corresponding binary operator.

```
infixl  1000   $$ ;               // sequence operator
infixr  1100   $ ;                // right-associative application
infixr  1200   , ;                // pair (tuple)
infix   1300   => ;               // key=>value pairs ("hash rocket")
infix   1400   .. ;               // arithmetic sequences
infixr  1500   || ;               // logical or (short-circuit)
infixr  1600   && ;               // logical and (short-circuit)
prefix  1700   ~ ;                // logical negation
infix   1800   < > <= >= == ~= ;  // relations
infix   1800   === ~== ;          // syntactic equality
infixr  1900   : ;                // list cons
infix   2000   +: <: ;            // complex numbers (cf. math.pure)
infixl  2100   << >> ;            // bit shifts
infixl  2200   + - or ;           // addition, bitwise or
infixl  2300   * / div mod and ;  // multiplication, bitwise and
infixl  2300   % ;                // exact division (cf. math.pure)
prefix  2400   not ;              // bitwise not
infixr  2500   ^ ;                // exponentiation
prefix  2600   # ;                // size operator
infixl  2700   ! !! ;             // indexing, slicing
infixr  2800   . ;                // function composition
prefix  2900   ' ;                // quote
postfix 3000   & ;                // thunk
```

2.1.2 Prelude Types

Some additional type symbols are provided which can be used as type tags on the left-hand side of equations, see *Type Tags* in the Pure Manual.

type number
type complex
type real
type rational
type integer
type bool
Additional number types.

These types are defined in a purely syntactic way, by checking the builtin-type or the constructor symbol of a number. Some semantic number types can be found in the `math` module, see Semantic Number Predicates and Types.

`integer` is the union of Pure's built-in integer types, i.e., it comprises all `int` and `bigint` values. `bool` is a subtype of `int` which denotes just the normalized truth values `0` and `1` (a.k.a. `false` and `true`).

`rational` and `complex` are the rational and complex types, while `real` is the union of the `double`, `integer` and `rational` types (i.e., anything that can represent a real number and be used for the real and imaginary parts of a `complex` number). Finally, `number` is the union of all numeric types, i.e., this type can be used to match any kind of number.

Note that the operations of the `rational` and `complex` types are actually defined in the `math` module which isn't part of the prelude, so you have to import this module in order to do computations with these types of values. However, the type tags and constructors for these types are defined in the prelude so that these kinds of values can be parsed and recognized without having the `math` module loaded.

The prelude also provides a subtype of the built-in `string` type which represents single-character strings:

type char
A single character string. This matches any string value of length 1.

Lists and tuples can be matched with the following types:

type list
type rlist
The list and "proper" (or "recursive") list types. Note that the former comprises both the empty list `[]` and all list nodes of the form `x:xs` (no matter whether the tail `xs` is a proper list value or not), whereas the latter only matches proper list values of the form `x1:...:xn:[]`. Thus the `list` type can be checked in O(1) time, while the `rlist` type is defined recursively and requires linear time (with respect to the size of the list) to be checked. This should be considered when deciding whether to use one or the other in a given situation; see *Type Rules* for further explanation.

type tuple

The type of all tuples, comprises the empty tuple `()` and all tuples `(x,xs)` with at least two members. This is analogous to the `list` type above, but no "proper" tuple type is needed here since any tuple of this form is always a proper tuple.

There are some other, more specialized types representing various kinds of applications, function objects and other named entities. These are useful, in particular, for the definition of higher-order functions and for performing symbolic manipulations on unevaluated symbolic terms.

type `appl`
This type represents all unevaluated function or constructor applications of the form `x y`. This comprises constructor terms and quoted or partial function applications.

type `function`
This type represents any term which may be called as a function. This may be a closure (global or local function, or a lambda function) which takes at least one argument, or a partial application of a closure to some arguments which is still "unsaturated", i.e., expects some further arguments to be "ready to go".

type `fun`
A named function object (global or local function, but not a partial application).

type `lambda`
An anonymous (lambda) function.

type `closure`
Any kind of function object (named function or lambda). This is the union of the `fun` and `lambda` types.

type `thunk`
This is a special kind of unevaluated parameterless function object used in lazy evaluation. See *Lazy Evaluation and Streams* in the Pure Manual.

type `var`
A free variable. This can be any kind of symbol that could in principle be bound to a value (excluding operator and nonfix symbols).

type `symbol`
Any kind of symbol (this also includes operator and nonfix symbols).

Corresponding type predicates are provided for all of the above, see Predicates. Some further types and predicates for matrices and records can be found under Matrix Inspection and Manipulation and Record Functions.

2.1.3 Basic Combinators

The prelude implements the following important function combinators.

f **`$`** g
f **`.`** g
Like in Haskell, these denote right-associative application and function composition.

They are also defined as macros so that saturated calls of them are eliminated automatically. Examples:

```
> foo $ bar 99;
foo (bar 99)
> (foo.bar) 99;
foo (bar 99)
```

id x

cst x y

These are the customary identity and constant combinators from the combinatorial calculus:

```
> map id (1..5);
[1,2,3,4,5]
> map (cst 0) (1..5);
[0,0,0,0,0]
```

void x

This combinator is basically equivalent to `cst ()`, but with the special twist that it is also defined as a macro optimizing the case of "throwaway" list and matrix comprehensions. This is useful if a comprehension is evaluated solely for its side effects. E.g.:

```
> using system;
> extern int rand();
> foo = void [printf "%d\n" rand | _ = 1..3];
> show foo
foo = do (\_ -> printf "%d\n" rand) (1..3);
> foo;
1714636915
1957747793
424238335
()
```

Note that the above list comprehension is actually implemented using `do` (instead of `map`, which would normally be the case), so that the intermediate list value of the comprehension is never constructed. This is described in more detail in section *Optimization Rules* of the Pure Manual.

In addition, the prelude also provides the following combinators adopted from Haskell:

flip f

Swaps arguments of a binary function `f`, e.g.:

```
> map (flip (/) 2) (1..3);
[0.5,1.0,1.5]
```

This combinator is also used by the compiler to implement right operator sections, which allows you to write the above simply as:

```
> map (/2) (1..3);
[0.5,1.0,1.5]
```

curry f

Turns a function `f` expecting a pair of values into a curried function of two arguments:

```
> using system;
> dowith (curry (printf "%d: %g\n")) (0..2) [0.0,2.718,3.14];
0: 0
1: 2.718
2: 3.14
()
```

uncurry f

The inverse of `curry`. Turns a curried function `f` expecting two arguments into a function processing a single pair argument:

```
> map (uncurry (*)) [(2,3),(4,5),(6,7)];
[6,20,42]
```

curry3 f
uncurry3 f

These work analogously, but are used to convert between ternary curried functions and functions operating on triples.

fix f

This is the (normal order) fixed point combinator which allows you to create recursive anonymous functions. It takes another function `f` as its argument and applies `f` to `fix f` itself:

```
> let fact = fix (\f n -> if n<=0 then 1 else n*f (n-1));
> map fact (1..5);
[1,2,6,24,120]
```

See Fixed point combinator at Wikipedia for an explanation of how this magic works. Just like in Haskell, `fix` can be used to produce least fixed points of arbitrary functions. For instance:

```
> fix (cst bar);
bar
> let xs = fix (1:);
> xs;
1:#<thunk 0x7fe537fe2f90>
> xs!!(0..10);
[1,1,1,1,1,1,1,1,1,1,1]
```

2.1.4 Lists and Tuples

The prelude defines the list and tuple constructors, as well as equality and inequality on these structures. It also provides a number of other useful basic operations on lists and tuples. These are all described below.

constructor []

constructor ()
Empty list and tuple.

constructor x **:** y
constructor x **,** y
List and tuple constructors. These are right-associative in Pure.

Lists are the usual right-recursive aggregates of the form `x:xs`, where `x` denotes the **head** and `xs` the **tail** of the list, pretty much the same as in Lisp or Prolog except that they use a Haskell-like syntax. In contrast to Haskell, list concatenation is denoted '+' (see below), and lists may contain an arbitrary mixture of arguments, i.e., they are fully polymorphic:

```
> 1:2:3:[];
[1,2,3]
> [1,2,3]+[u,v,w]+[3.14];
[1,2,3,u,v,w,3.14]
```

Lists are **eager** in Pure by default, but they can also be made **lazy** (in the latter case they are also called **streams**). This is accomplished by turning the tail of a list into a "thunk" (a.k.a. "future") which defers evaluation until the list tail is actually needed, see section *Lazy Evaluation and Streams* in the Pure Manual. For instance, an infinite arithmetic sequence (see below) will always produce a list with a thunked tail:

```
> 1:3..inf;
1:#<thunk 0x7f696cd2dbd8>
```

Pure also distinguishes **proper** and **improper** lists. The former are always terminated by an empty list in the final tail and can thus be written using the conventional `[x1,x2,...,xn]` syntax:

```
> 1:2:3:[];
[1,2,3]
```

In contrast, improper lists are terminated with a non-list value and can only be represented using the ':' operator:

```
> 1:2:3;
1:2:3
```

These aren't of much use as ordinary list values, but are frequently encountered as patterns on the left-hand side of an equation, where the final tail is usually a variable. Also note that technically, a lazy list is also an improper list (although it may expand to a proper list value as it is traversed).

Tuples work in a similar fashion, but with the special twist that the pairing constructor ',' is associative (it always produces right-recursive pairs) and '()' acts as a neutral element on these constructs, so that ',' and '()' define a complete monoid structure. Note that this means that ',' is actually a "constructor with equations" since it obeys the laws `(x,y),z == x,(y,z)` and `(),x == x,() == x`. Also note that there isn't a separate operation for concatenating tuples, since the pairing operator already does this:

```
> (1,2,3),(10,9,8);
1,2,3,10,9,8
> (),(a,b,c);
a,b,c
> (a,b,c),();
a,b,c
```

This also implies that tuples are always flat in Pure and can't be nested; if you need this, you should use lists instead. Also, tuples are always eager in Pure.

Some important basic operations on lists and tuples are listed below.

x **+** y

List concatenation. This non-destructively appends the elements of y to x.

```
> [1,2,3]+[u,v,w];
[1,2,3,u,v,w]
```

Note that this operation in fact just recurses into x and replaces the empty list marking the "end" of x with y, as if defined by the following equations (however, the prelude actually defines this operation in a tail-recursive fashion):

```
[] + ys = ys;
(x:xs) + ys = x : xs+ys;
```

To make this work, both operands should be proper lists, otherwise you may get somewhat surprising (but correct) improper list results like the following:

```
> [1,2,3]+99;
1:2:3:99
> (1:2:3)+33;
1:2:36
```

This happens because Pure is dynamically typed and places no limits on ad hoc polymorphism. Note that the latter result is due to the fact that '+' also denotes the addition of numbers, and the improper tail of the first operand is a number in this case, as is the second operand. Otherwise you might have got an unreduced instance of the '+' operator instead.

x **==** y

x **~=** y

Equality and inequality of lists and tuples. These compare two lists or tuples by recursively comparing their members, so '==' must be defined on the list or tuple members if you want to use these operations. Also note that these operations are inherently eager, so applying them to two infinite lists may take an infinite amount of time.

```
> reverse [a,b,c] == [c,b,a];
1
> (a,b,c) == ();
0
```

x

List and tuple size. This operation counts the number of elements in a list or tuple:

```
> #[a,b,c];
3
> #(a,b,c);
3
```

Please note that for obvious reasons this operation is inherently eager, so trying to compute the size of an infinite list will take forever.

x `!` i

Indexing of lists and tuples is always zero-based (i.e., indices run from `0` to `#x-1`), and an exception will be raised if the index is out of bounds:

```
> [1,2,3]!2;
3
> [1,2,3]!4;
<stdin>, line 34: unhandled exception 'out_of_bounds' while evaluating
'[1,2,3]!4'
```

x `!!` is

The slicing operation takes a list or tuple and a list of indices and returns the list or tuple of the corresponding elements, respectively. Indices which are out of the valid range are silently ignored:

```
> (1..5)!!(3..10);
[4,5]
> (1,2,3,4,5)!!(3..10);
4,5
```

The case of contiguous index ranges, as shown above, is optimized so that it always works in linear time, see Slicing below for details. But indices can actually be specified in any order, so that you can retrieve any permutation of the members, also with duplicates. E.g.:

```
> (1..5)!![2,4,4,1];
[3,5,5,2]
```

This is less efficient than the case of contiguous index ranges, because it requires repeated traversals of the list for each index. For larger lists you should hence use vectors or matrices instead, to avoid the quadratic complexity.

x `..` y

Arithmetic sequences. Note that the Pure syntax differs from Haskell in that there are no brackets around the construct and a step width is indicated by specifying the first two elements as `x:y` instead of `x,y`.

```
> 1..5;
[1,2,3,4,5]
> 1:3..11;
[1,3,5,7,9,11]
```

To prevent unwanted artifacts due to rounding errors, the upper bound in a floating point sequence is always rounded to the nearest grid point:

```
> 0.0:0.1..0.29;
[0.0,0.1,0.2,0.3]
> 0.0:0.1..0.31;
[0.0,0.1,0.2,0.3]
```

Last but not least, you can specify infinite sequences with an infinite upper bound (`inf` or `-inf`):

```
> 1:3..inf;
1:#<thunk 0x7f696cd2dbd8>
> -1:-3..-inf;
-1:#<thunk 0x7f696cd2fde8>
```

The lower bounds of an arithmetic sequence must always be finite.

`null` x
: Test for the empty list and tuple.

```
> null [];
1
> null (a,b,c);
0
```

`reverse` x
: Reverse a list or tuple.

```
> reverse (1..5);
[5,4,3,2,1]
> reverse (a,b,c);
(c,b,a)
```

In addition, the prelude provides the following conversion operations.

`list` x
`tuple` x
: Convert between (finite) lists and tuples.

```
> tuple (1..5);
1,2,3,4,5
> list (a,b,c);
[a,b,c]
```

The `list` function can be used to turn a finite lazy list into an eager one:

```
> list $ take 10 (-1:-3..-inf);
[-1,-3,-5,-7,-9,-11,-13,-15,-17,-19]
```

You can also achieve the same effect somewhat more conveniently by slicing a finite part from a stream:

```
> (-1:-3..-inf)!!(0..9);
[-1,-3,-5,-7,-9,-11,-13,-15,-17,-19]
```

Conversely, it is also possible to convert an (eager) list to a lazy one (a stream).

stream x
: Convert a list to a stream.

```
> stream (1..10);
1:#<thunk 0x7fe537fe2b58>
```

This might appear a bit useless at first sight, since all elements of the stream are in fact already known. However, this operation then allows you to apply other functions to the list and have them evaluated in a lazy fashion.

2.1.5 Slicing

Indexing and slicing are actually fairly general operations in Pure which are used not only in the context of lists and tuples, but for any type of container data structure which can be "indexed" in some way. Other examples in the standard library are the `array` and `dict` containers.

The prelude therefore implements slicing in a generic way, so that it works with any kind of container data structure which defines `'!'` in such a manner that it throws an exception when the index is out of bounds. It also works with any kind of index container that implements the `catmap` operation.

The prelude also optimizes the case of contiguous integer ranges so that slices like `xs!!(i..j)` are computed in linear time if possible. This works, in particular, with lists, strings and matrices.

Moreover, the prelude includes some optimization rules and corresponding helper functions to optimize the most common cases at compile time, so that the index range is never actually constructed. To these ends, the slicing expression `xs!!(i..j)` is translated to a call `subseq xs i j` of the special `subseq` function:

subseq x i j
: If `x` is a list, matrix or string, and `i` and `j` are int values, compute the slice `xs!!(i..j)` in the most efficient manner possible. This generally avoids constructing the index list `i..j`. Otherwise `i..j` is computed and `subseq` falls back to the `slice` function below to compute the slice in the usual way.

slice x ys
: Compute the slice `x!!ys` using the standard slicing operation, without any special compile time tricks. (Runtime optimizations are still applied if possible.)

You can readily see the effects of this optimization by running the slicing operator against `slice`:

```
> let xs = 1..1000000;
> stats -m
> #slice xs (100000..299990);
199991
0.34s, 999957 cells
> #xs!!(100000..299990);
199991
0.14s, 399984 cells
```

Even more drastic improvements in both running time and memory usage can be seen in the case of matrix slices:

```
> let x = rowvector xs;
> #slice x (100000..299990);
199991
0.19s, 599990 cells
> #x!!(100000..299990);
199991
0s, 10 cells
```

2.1.6 Hash Pairs

The prelude provides another special kind of pairs called "hash pairs", which take the form `key=>value`. These are used in various contexts to denote key-value associations. The only operations on hash pairs provided by the prelude are equality testing (which recursively compares the components) and the functions `key` and `val`:

constructor x **=>** y
: The hash pair constructor, also known as the "hash rocket".

x **==** y
x **~=** y
: Equality and inequality of hash pairs.

```
> ("foo"=>99) == ("bar"=>99);
0
```

key (x=>y)
val (x=>y)
: Extract the components of a hash pair.

```
> key ("foo"=>99), val ("foo"=>99);
"foo",99
```

Note that in difference to the tuple operator '`,`', the hash rocket '`=>`' is non-associative, so nested applications *must* be parenthesized, and `(x=>y)=>z` is generally *not* the same as `x=>(y=>z)`. Also note that '`,`' has lower precedence than '`=>`', so to include a tuple as key or value in a hash pair, the tuple must be parenthesized, as in `"foo"=>(1,2)` (whereas `"foo"=>1,2` denotes a tuple whose first element happens to be a hash pair).

2.1.7 List Functions

This mostly comes straight from the Q prelude which in turn was based on the first edition of the Bird/Wadler book, and is very similar to what you can find in the Haskell prelude. Some functions have slightly different names, though, and of course everything is typed dynamically.

Common List Functions

any p xs
: test whether the predicate `p` holds for any of the members of `xs`

all p xs
: test whether the predicate `p` holds for all of the members of `xs`

cat xs
: concatenate a list of lists

catmap f xs
: convenience function which combines `cat` and `map`; this is also used to implement list comprehensions

do f xs
: apply `f` to all members of `xs`, like `map`, but throw away all intermediate results and return `()`

drop n xs
: remove `n` elements from the front of `xs`

dropwhile p xs
: remove elements from the front of `xs` while the predicate `p` is satisfied

filter p xs
: return the list of all members of `xs` satisfying the predicate `p`

foldl f a xs
: accumulate the binary function `f` over all members of `xs`, starting from the initial value `a` and working from the front of the list towards its end

foldl1 f xs
: accumulate the binary function `f` over all members of `xs`, starting from the value `head xs` and working from the front of the list towards its end; `xs` must be nonempty

foldr f a xs
: accumulate the binary function `f` over all members of `xs`, starting from the initial value `a` and working from the end of the list towards its front

foldr1 f xs
: accumulate the binary function `f` over all members of `xs`, starting from the value `last xs` and working from the end of the list towards its front; `xs` must be nonempty

head xs
: return the first element of `xs`; `xs` must be nonempty

index xs x
: search for an occurrence of `x` in `xs` and return the index of the first occurrence, if any, `-1` otherwise

  Note: This uses equality `==` to decide whether a member of `xs` is an occurrence of `x`, so `==` must have an appropriate definition on the list members.

init xs
: return all but the last element of `xs`; `xs` must be nonempty

last xs
: return the last element of `xs`; `xs` must be nonempty

listmap f xs
: convenience function which works like `map`, but also deals with matrix and string arguments while ensuring that the result is always a list; this is primarily used to implement list comprehensions

map f xs
: apply `f` to each member of `xs`

scanl f a xs
: accumulate the binary function `f` over all members of `xs`, as with `foldl`, but return all intermediate results as a list

scanl1 f xs
: accumulate the binary function `f` over all members of `xs`, as with `foldl1`, but return all intermediate results as a list

scanr f a xs
: accumulate the binary function `f` over all members of `xs`, as with `foldr`, but return all intermediate results as a list

scanr1 f xs
: accumulate the binary function `f` over all members of `xs`, as with `foldr1`, but return all intermediate results as a list

sort p xs
: Sorts the elements of the list `xs` in ascending order according to the given predicate `p`, using the C `qsort` function. The predicate `p` is invoked with two arguments and should return a truth value indicating whether the first argument is "less than" the second. (An exception is raised if the result of a comparison is not a machine integer.)

```
> sort (>) (1..10);
[10,9,8,7,6,5,4,3,2,1]
> sort (<) ans;
[1,2,3,4,5,6,7,8,9,10]
```

tail xs
: return all but the first element of `xs`; `xs` must be nonempty

take n xs
: take `n` elements from the front of `xs`

takewhile p xs
: take elements from the front of `xs` while the predicate `p` is satisfied

List Generators

Some useful (infinite) list generators, as well as some finite (and eager) variations of these. The latter work like a combination of `take` or `takewhile` and the former, but are implemented directly for better efficiency.

cycle xs
: cycles through the elements of the nonempty list `xs`, ad infinitum

cyclen n xs
: eager version of `cycle`, returns the first `n` elements of `cycle xs`

iterate f x
: returns the stream containing `x`, `f x`, `f (f x)`, etc., ad infinitum

iteraten n f x
: eager version of `iterate`, returns the first `n` elements of `iterate f x`

iterwhile p f x
: another eager version of `iterate`, returns the list of all elements from the front of `iterate f x` for which the predicate `p` holds

repeat x
: returns an infinite stream of xs

repeatn n x
: eager version of `repeat`, returns a list with `n` xs

Zip and Friends

unzip xys
: takes a list of pairs to a pair of lists of corresponding elements

unzip3 xyzs
: `unzip` with triples

zip xs ys
: return the list of corresponding pairs `(x,y)` where `x` runs through the elements of `xs` and `y` runs through the elements of `ys`

zip3 xs ys zs
: `zip` with three lists, returns a list of triples

zipwith f xs ys
: apply the binary function `f` to corresponding elements of `xs` and `ys`

zipwith3 f xs ys zs
: apply the ternary function `f` to corresponding elements of `xs`, `ys` and `zs`

Pure also has the following variations of `zipwith` and `zipwith3` which throw away all intermediate results and return the empty tuple `()`. That is, these work like `do` but pull arguments from two or three lists, respectively:

dowith f xs ys
: apply the binary function `f` to corresponding elements of `xs` and `ys`, return `()`

dowith3 f xs ys zs
: apply the ternary function `f` to corresponding elements of `xs`, `ys` and `zs`, return `()`

2.1.8 String Functions

Pure strings are null-terminated character strings encoded in UTF-8, see the Pure Manual for details. The prelude provides various operations on strings, including a complete set of list-like operations, so that strings can be used mostly as if they were lists, although they are really implemented as C character arrays for reasons of efficiency. Pure also has some powerful operations to convert between Pure expressions and their string representation, see Eval and Friends for those.

Basic String Functions

s **+** t
s **!** i
s **!!** is
: String concatenation, indexing and slicing works just like with lists:

```
> "abc"+"xyz";
"abcxyz"
> let s = "The quick brown fox jumps over the lazy dog.";
> s!5;
"u"
> s!!(20..24);
"jumps"
```

null s
s
: Checking for empty strings and determining the size of a string also works as expected:

```
> null "";
1
> null s;
0
> #s;
44
```

s **==** t

s **~=** t
s **<=** t
s **>=** t
s **<** t
s **>** t

String equality and comparisons. This employs the usual lexicographic order based on the (UTF-8) character codes.

```
> "awe">"awesome";
0
> "foo">="bar";
1
> "foo"=="bar";
0
```

You can search for the location of a substring in a string, and extract a substring of a given length:

index s u

Returns the (zero-based) index of the first occurrence of the substring `u` in `s`, or -1 if `u` is not found in `s`.

substr s i n

Extracts a substring of (at most) `n` characters at position `i` in `s`. This takes care of all corner cases, adjusting index and number of characters so that the index range stays confined to the source string.

Example:

```
> index s "jumps";
20
> substr s 20 10;
"jumps over"
```

Note that Pure doesn't have a separate type for individual characters. Instead, these are represented as strings `c` containing exactly one (UTF-8) character (i.e., `#c==1`). It is possible to convert such single character strings to the corresponding integer character codes, and vice versa:

ord c

Ordinal number of a single character string `c`. This is the character's code point in the Unicode character set.

chr n

Converts an integer back to the character with the corresponding code point.

In addition, the usual character arithmetic works, including arithmetic sequences of characters, so that you can write stuff like the following:

```
> "a"-"A";
32
> "u"-32;
```

```
"U"
> "a".."k";
["a","b","c","d","e","f","g","h","i","j","k"]
```

For convenience, the prelude provides the following functions to convert between strings and lists (or other aggregates) of characters.

chars s
list s
: Convert a string `s` to a list of characters.

tuple s
matrix s
: Convert a string `s` to a tuple or (symbolic) matrix of characters, respectively.

strcat xs
: Concatenate a list `xs` of strings (in particular, this converts a list of characters back to a string).

string xs
: Convert a list, tuple or (symbolic) matrix of strings to a string. In the case of a list, this is synonymous with `strcat`, but it also works with the other types of aggregates.

For instance:

```
> list "abc";
["a","b","c"]
> string ("a".."z");
"abcdefghijklmnopqrstuvwxyz"
```

The following functions are provided to deal with strings of "tokens" separated by a given delimiter string.

split delim s
: Splits `s` into a list of substrings delimited by `delim`.

join delim xs
: Joins the list of strings `xs` to a single string, interpolating the given `delim` string.

Example:

```
> let xs = split " " s; xs;
["The","quick","brown","fox","jumps","over","the","lazy","dog."]
> join ":" xs;
"The:quick:brown:fox:jumps:over:the:lazy:dog."
```

We mention in passing here that more elaborate string matching, splitting and replacement operations based on regular expressions are provided by the system module, see Regex Matching.

If that isn't enough already, most generic list operations carry over to strings in the obvious way, treating the string like a list of characters. (Polymorphic operations such as `map`, which aren't guaranteed to yield string results under all circumstances, will actually return lists in

that case, so you might have to apply `string` explicitly to convert these back to a string.) For instance:

```
> filter (>="k") s;
"qukrownoxumpsovrtlzyo"
> string $ map pred "ibm";
"hal"
```

List comprehensions can draw values from strings, too:

```
> string [x+1 | x="HAL"];
"IBM"
```

Low-Level Operations

The following routines are provided by the runtime to turn raw C `char*` pointers (also called **byte strings** in Pure parlance, to distinguish them from Pure's "cooked" UTF-8 string values) into corresponding Pure strings. Normally you don't have to worry about this, because the C interface already takes care of the necessary marshalling, but in some low-level code these operations are useful. Also note that here and in the following, the `cstring` routines also convert the string between the system encoding and Pure's internal UTF-8 representation.

string s
cstring s
: Convert a pointer `s` to a Pure string. `s` must point to a null-terminated C string. These routines take ownership of the original string value, assuming it to be malloced, so you should only use these for C strings which are specifically intended to be freed by the user.

string_dup s
cstring_dup s
: Convert a pointer `s` to a Pure string. Like above, but these functions take a copy of the string, leaving the original C string untouched.

The reverse transformations are also provided. These take a Pure string to a byte string (raw `char*`).

byte_string s
byte_cstring s
: Construct a byte string from a Pure string `s`. The result is a raw pointer object pointing to the converted string. The original Pure string is always copied (and, in the case of `byte_cstring`, converted to the system encoding). The resulting byte string is a malloced pointer which can be used like a C `char*`, and has to be freed explicitly by the caller when no longer needed.

It is also possible to convert Pure string lists or symbolic vectors of strings to byte string vectors and vice versa. These are useful if you need to pass an `argv`-like string vector (i.e., a `char**` or `char*[]`) to C routines. The computed C vectors are malloced pointers which have an extra `NULL` pointer as the last entry, and should thus be usable for almost any purpose which requires such a string vector in C. They also take care of garbage-collecting

themselves. The original string data is always copied. As usual, the `cstring` variants do automatic conversions to the system encoding.

byte_string_pointer xs
byte_cstring_pointer xs
Convert a list or vector of Pure strings to a C `char**`.

string_list n p
cstring_list n p
Convert a C `char**` to a list of Pure strings.

string_vector n p
cstring_vector n p
Convert a C `char**` to a symbolic vector of Pure strings.

Note that the back conversions take an additional first argument which denotes the number of strings to retrieve. If you know that the vector is `NULL`-terminated then this can also be an infinite value (`inf`) in which case the number of elements will be figured out automatically. Processing always stops at the first `NULL` pointer encountered.

Also note that, as of version 0.45, Pure has built-in support for passing `argv`-style vectors as arguments by means of the `char**` and `void**` pointer types. However, the operations provided here are more general in that they allow you to both encode and decode such values in an explicit fashion. This is useful, e.g., for operations like `getopt` which may mutate the given `char**` vector.

If you have `getopt` in your C library, you can try the following example. First enter these definitions:

```
extern int getopt(int argc, char **argv, char *optstring);
optind = get_int $ addr "optind";
optarg = cstring_dup $ get_pointer $ addr "optarg";
```

Now let's run `getopt` on a byte string vector constructed from an argument vector (which includes the "program name" in the first element):

```
> let args = byte_cstring_pointer {"progname","boo","-n","-tfoo","bar"};
> getopt 5 args "nt:", optarg;
110,#<pointer 0>
> getopt 5 args "nt:", optarg;
116,"foo"
> getopt 5 args "nt:", optarg;
-1,#<pointer 0>
```

Note that 110 and 116 are the character codes of the option characters `n` and `t`, where the latter option takes an argument, as returned by `optarg`. Finally, `getopt` returns -1 to indicate that there are no more options, and we can retrieve the current `optindex` value and the mutated argument vector to see which non-option arguments remain to be processed, as follows:

```
> optind, cstring_vector 5 args;
3,{"progname","-n","-tfoo","boo","bar"}
```

It is now an easy exercise to design your own high-level wrapper around `getopt` to process command line arguments in Pure. However, this isn't really necessary since the Pure library already offers such an operation which doesn't rely on any special system functions, see Option Parsing in the System Interface section.

2.1.9 Matrix Functions

Matrices are provided as an alternative to the list and tuple aggregates which provide contant time access to their members and are tailored for use in numeric computations.

x
dim x
: Determine the size of a matrix (number of elements) and its dimensions (number of rows and columns).

```
> let x = {1,2,3;4,5,6}; #x;
6
> dim x;
2,3
```

null
: Check for empty matrices. Note that there are various kinds of these, as a matrix may have zero rows or columns, or both.

x **==** y
x **~=** y
: Matrix equality and inequality. These check the dimensions and the matrix elements for equality:

```
> x == transpose x;
0
```

x **!** i
x **!!** is
: Indexing and slicing.

Indexing and slicing employ the standard Pure operators '`!`' and '`!!`'. They work pretty much like in MATLAB and Octave, but note that Pure matrices are in row-major order and the indices are zero-based. It is possible to access elements with a one-dimensional index (in row-major oder):

```
> x!3;
4
```

Or you can specify a pair of row and column index:

```
> x!(1,0);
4
```

Slicing works accordingly. You can either specify a list of (one- or two-dimensional) indices, in which case the result is always a row vector:

```
> x!!(2..5);
{3,4,5,6}
```

Or you can specify a pair of row and column index lists:

```
> x!!(0..1,1..2);
{2,3;5,6}
```

The following abbreviations are provided to grab a slice from a row or column:

```
> x!!(1,1..2);
{5,6}
> x!!(0..1,1);
{2;5}
```

As in the case of lists, matrix slices are optimized to handle cases with contiguous index ranges in an efficient manner, see Slicing for details. To these ends, the helper functions `subseq` and `subseq2` are defined to handle the necessary compile time optimizations.

Most of the generic list operations are implemented on matrices as well, see Common List Functions. Hence operations like `map` and `zipwith` work as expected:

```
> map succ {1,2,3;4,5,6};
{2,3,4;5,6,7}
> zipwith (+) {1,2,3;4,5,6} {1,0,1;0,2,0};
{2,2,4;4,7,6}
```

The matrix module also provides a bunch of other specialized matrix operations, including all the necessary operations for matrix comprehensions. We briefly summarize the most important operations below; please refer to matrices.pure for all the gory details. Also make sure you check *Matrices and Vectors* in the Pure Manual for some more examples, and the Record Functions section for an implementation of records using symbolic vectors.

Matrix Construction and Conversions

matrix xs

This function converts a list or tuple to a corresponding matrix. `matrix` also turns a list of lists or matrices specifying the rows of the matrix to the corresponding rectangular matrix; otherwise, the result is a row vector. (In the former case, `matrix` may throw a `bad_matrix_value` exception in case of dimension mismatch, with the offending submatrix as argument.)

```
> matrix [1,2,3];
{1,2,3}
> matrix [[1,2,3],[4,5,6]];
{1,2,3;4,5,6}
```

rowvector xs
colvector xs

vector xs

The rowvector and colvector functions work in a similar fashion, but expect a list, tuple or matrix of elements and always return a row or column vector, respectively (i.e., a $1 \times n$ or $n \times 1$ matrix, where n is the size of the converted aggregate). Also, the vector function is a synonym for rowvector. These functions can also be used to create recursive (symbolic) matrix structures of arbitrary depth, which provide a nested array data structure with efficient (constant time) element access.

```
> rowvector [1,2,3];
{1,2,3}
> colvector [1,2,3];
{1;2;3}
> vector [rowvector [1,2,3],colvector [4,5,6]];
{{1,2,3},{4;5;6}}
```

Note that for convenience, there's also an alternative syntax for entering nested vectors more easily, see the description of the *non-splicing vector brackets* below for details.

rowvectorseq x y step
colvectorseq x y step
vectorseq x y step

With these functions you can create a row or column vector from an arithmetic sequence. Again, vectorseq is provided as a synonym for rowvectorseq. These operations are optimized for the case of int and double ranges.

```
> rowvectorseq 0 10 1;
{0,1,2,3,4,5,6,7,8,9,10}
> colvectorseq 0 10 1;
{0;1;2;3;4;5;6;7;8;9;10}
> vectorseq 0.0 0.9 0.1;
{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}
```

The prelude also contains some optimization rules which translate calls to vector et al on arithmetic sequences to the corresponding calls to vectorseq et al, such as:

```
def vector (n1:n2..m) = vectorseq n1 m (n2-n1);
def vector (n..m) = vectorseq n m 1;
```

Example:

```
> foo = vector (1..10);
> bar = vector (0.0:0.1..0.9);
> show foo bar
bar = vectorseq 0.0 0.9 0.1;
foo = vectorseq 1 10 1;
> foo; bar;
{1,2,3,4,5,6,7,8,9,10}
{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}
```

Please note that these optimization rules assume that basic arithmetic works with the involved elements, which may give you trouble if you try to use vector et al with

exotic kinds of user-defined arithmetic sequences. To disable them, simply run the interpreter with the option `--disable vectorseq-opt`.

dmatrix xs
cmatrix xs
imatrix xs
smatrix xs

These functions convert a list or matrix to a matrix of the corresponding type (integer, double, complex or symbolic). If the input is a list, the result is always a row vector; this is usually faster than the `matrix` and `vector` operations, but requires that the elements already are of the appropriate type.

```
> imatrix [1,2,3];
{1,2,3}
> dmatrix {1,2,3;4,5,6};
{1.0,2.0,3.0;4.0,5.0,6.0}
```

The `dmatrix`, `cmatrix` and `imatrix` functions can also be invoked with either an int `n` or a pair `(n,m)` of ints as argument, in which case they construct a zero rowvector or matrix with the corresponding dimensions.

```
> imatrix 3;
{0,0,0}
> imatrix (2,3);
{0,0,0;0,0,0}
```

list x
list2 x
tuple x

These convert a matrix back to a flat list or tuple. The `list2` function converts a matrix to a list of lists (one sublist for each row of the matrix).

```
> tuple {1,2,3;4,5,6};
1,2,3,4,5,6
> list {1,2,3;4,5,6};
[1,2,3,4,5,6]
> list2 {1,2,3;4,5,6};
[[1,2,3],[4,5,6]]
> list2 {1,2,3};
[[1,2,3]]
```

In addition, the following special syntax is provided as a shorthand notation for nested vector structures:

macro {| x, y, z, ... **|}**

Non-splicing vector brackets. These work like `{x,y,z,...}`, but unlike these they will *not* splice submatrices in the arguments `x,y,z,...` So they work a bit like quoted vectors `'{x,y,z,...}`, but the arguments `x,y,z,...` will be evaluated as usual.

The non-splicing vector brackets provide a convenient shorthand to enter symbolic vector values which may contain other vectors or matrices as components. For instance, note how the ordinary matrix brackets combine the column subvectors in the first example below to

a 3x2 matrix, while the non-splicing brackets in the second example create a 1x2 row vector with the column vectors as members instead:

```
> {{1;2;3},{4;5;6}};
{1,4;2,5;3,6}
> {|{1;2;3},{4;5;6}|};
{{1;2;3},{4;5;6}}
```

The second example works like a quoted matrix expression such as `'{{1;2;3},{4;5;6}}`, but the non-splicing brackets also evaluate their arguments:

```
> '{vector (1..3),vector (4..6)};
{vector (1..3),vector (4..6)}
> {|vector (1..3),vector (4..6)|};
{{1,2,3},{4,5,6}}
```

The `{| |}` brackets can be nested. Examples:

```
> {|1,{|vector (1..5),2*3|},{}|};
{1,{{1,2,3,4,5},6},{}}
> {|{|{1,2}|},{|{3,4}|}|};
{{{1,2}},{{3,4}}}
```

Also note that the `{| |}` brackets only produce row vectors, but you can just transpose the result if you need a column vector instead:

```
> transpose {|{1;2;3},{4;5;6}|};
{{1;2;3};{4;5;6}}
```

Finally, note that the notation `{| |}` without any arguments is not supported, simply write `{}` for the empty vector instead.

Matrix Inspection and Manipulation

type dmatrix
type cmatrix
type imatrix
type smatrix
type nmatrix

Convenience types for the different subtypes of matrices (double, complex, int, symbolic and numeric, i.e., non-symbolic). These can be used as type tags on the left-hand side of equations to match specific types of matrices.

dmatrixp x
cmatrixp x
imatrixp x
smatrixp x
nmatrixp x

Corresponding predicates to check for different kinds of matrices.

vectorp x

rowvectorp x
colvectorp x
Check for different kinds of vectors (these are just matrices with one row or column).

stride x
The stride of a matrix denotes the real row size of the underlying C array, see the description of the `pack` function below for further details. There's little use for this value in Pure, but it may be needed when interfacing to C.

subseq x i j
subseq2 x i j k l
Helper functions to optimize matrix slices, see Slicing for details. `subseq2` is a special version of `subseq` which is used to optimize the case of 2-dimensional matrix slices `xs!!(i..j,k..l)`.

row x i
col x i
Extract the `i`th row or column of a matrix.

rows x
cols x
Return the list of all rows or columns of a matrix.

diag x
subdiag x k
supdiag x k
Extract (sub-,super-) diagonals from a matrix. Sub- and super-diagonals for `k=0` return the main diagonal. Indices for sub- and super-diagonals can also be negative, in which case the corresponding super- or sub-diagonal is returned instead. In each case the result is a row vector.

submat x (i,j) (n,m)
Extract a submatrix of a given size at a given offset. The result shares the underlying storage with the input matrix (i.e., matrix elements are *not* copied) and so this is a comparatively cheap operation.

rowcat xs
colcat xs
Construct matrices from lists of rows and columns. These take either scalars or submatrices as inputs; corresponding dimensions must match. `rowcat` combines submatrices vertically, like `{x;y}`; `colcat` combines them horizontally, like `{x,y}`. Note: Like the built-in matrix constructs, these operations may throw a `bad_matrix_value` exception in case of dimension mismatch.

matcat xs
Construct a matrix from a (symbolic) matrix of other matrices and/or scalars. This works like a combination of `rowcat` and `colcat`, but draws its input from a matrix instead of a list of matrices, and preserves the overall layout of the "host" matrix. The net effect is that the host matrix is flattened out. If all elements of the input matrix are scalars already, the input matrix is returned unchanged.

rowcatmap f xs
colcatmap f xs
rowmap f xs
colmap f xs
: Various combinations of `rowcat`, `colcat` and `map`. These are used, in particular, for implementing matrix comprehensions.

diagmat x
subdiagmat x k
supdiagmat x k
: Create a (sub-,super-) diagonal matrix from a row vector `x` of size `n`. The result is always a square matrix with dimension `(n+k,n+k)`, which is of the same matrix type (double, complex, int, symbolic) as the input and has the elements of the vector on its `k`th sub- or super-diagonal, with all other elements zero. A negative value for `k` turns a sub- into a super-diagonal matrix and vice versa.

re x
im x
conj x
: Extract the real and imaginary parts and compute the conjugate of a numeric matrix.

pack x
packed x
: Pack a matrix. This creates a copy of the matrix which has the data in contiguous storage. It also frees up extra memory if the matrix was created as a slice from a bigger matrix (see `submat` above) which has since gone the way of the dodo. The `packed` predicate can be used to verify whether a matrix is already packed. Note that even if a matrix is already packed, `pack` will make a copy of it anyway, so `pack` also provides a quick way to copy a matrix, e.g., if you want to pass it as an input/output parameter to a GSL routine.

redim (n,m) x
redim n x
: Change the dimensions of a matrix without changing its size. The total number of elements must match that of the input matrix. Reuses the underlying storage of the input matrix if possible (i.e., if the matrix is `packed`). You can also redim a matrix to a given row size `n`. In this case the row size must divide the total size of the matrix.

sort p x
: Sorts the elements of a matrix (non-destructively, i.e., without changing the original matrix) according to the given predicate, using the C `qsort` function. This works exactly the same as with lists (see Common List Functions), except that it takes and returns a matrix instead of a list. Note that the function sorts *all* elements of the matrix in one go (regardless of the dimensions), as if the matrix was a single big vector. The result matrix has the same dimensions as the input matrix. Example:

```
> sort (<) {10,9;8,7;6,5};
{5,6;7,8;9,10}
```

If you'd like to sort the individual rows instead, you can do that as follows:

```
> sort_rows p = rowcat . map (sort p) . rows;
> sort_rows (<) {10,9;8,7;6,5};
{9,10;7,8;5,6}
```

Likewise, to sort the columns of a matrix:

```
> sort_cols p = colcat . map (sort p) . cols;
> sort_cols (<) {10,9;8,7;6,5};
{6,5;8,7;10,9}
```

Also note that the pure-gsl module provides an interface to the GSL routines for sorting numeric (int and double) vectors using the standard order. These will usually be much faster than `sort`, whereas `sort` is more flexible in that it also allows you to sort symbolic matrices and to choose the order predicate.

`transpose` x
: Transpose a matrix. Example:

```
> transpose {1,2,3;4,5,6};
{1,4;2,5;3,6}
```

`rowrev` x
`colrev` x
`reverse` x
: Reverse a matrix. `rowrev` reverses the rows, `colrev` the columns, `reverse` both dimensions.

Pointers and Matrices

Last but not least, the matrix module also offers a bunch of low-level operations for converting between matrices and raw pointers. These are typically used to shovel around massive amounts of numeric data between Pure and external C routines, when performance and throughput is an important consideration (e.g., graphics, video and audio applications). The usual caveats concerning direct pointer manipulations apply.

`pointer` x
: Get a pointer to the underlying C array of a matrix. The data is *not* copied. Hence you have to be careful when passing such a pointer to C functions if the underlying data is non-contiguous; when in doubt, first use the `pack` function to place the data in contiguous storage, or use one of the matrix-pointer conversion routines below.

`double_pointer` p x
`float_pointer` p x
`complex_pointer` p x
`complex_float_pointer` p x
`int_pointer` p x
`short_pointer` p x
`byte_pointer` p x
: These operations copy the contents of a matrix to a given pointer and return that

pointer, converting to the target data type on the fly if necessary. The given pointer may also be `NULL`, in which case suitable memory is malloced and returned; otherwise the caller must ensure that the memory pointed to by `p` is big enough for the contents of the given matrix.

`double_matrix` (n,m) p
`float_matrix` (n,m) p
`complex_matrix` (n,m) p
`complex_float_matrix` (n,m) p
`int_matrix` (n,m) p
`short_matrix` (n,m) p
`byte_matrix` (n,m) p

These functions allow you to create a numeric matrix from a pointer, copying the data and converting it from the source type on the fly if necessary. The source pointer `p` may also be `NULL`, in which case the new matrix is filled with zeros instead. Otherwise the caller must ensure that the pointer points to properly initialized memory big enough for the requested dimensions. The given dimension may also be just an integer `n` if a row vector is to be created.

`double_matrix_view` (n,m) p
`complex_matrix_view` (n,m) p
`int_matrix_view` (n,m) p

These operations can be used to create a numeric matrix view of existing data, without copying the data. The data must be double, complex or int, the pointer must not be `NULL` and the caller must also ensure that the memory persists for the entire lifetime of the matrix object. The given dimension may also be just an integer `n` if a row vector view is to be created.

2.1.10 Record Functions

As of Pure 0.41, the prelude also provides a basic record data structure, implemented as symbolic vectors of `key=>value` pairs which support a few dictionary-like operations such as `member`, `insert` and indexing. Records may be represented as row, column or empty vectors (i.e., the number of rows or columns must be zero or one). They must be symbolic matrices consisting only of "hash pairs" `key=>value`, where the keys can be either symbols or strings. The values can be any kind of Pure data; in particular, they may themselves be records, so records can be nested.

The following operations are provided. Please note that all updates of record members are non-destructive and thus involve copying, which takes linear time (and space) and thus might be slow for large record values; if this is a problem then you should use dictionaries instead (cf. Dictionaries). Or you can create mutable records by using expression references (cf. Expression References) as values, which allow you to modify the data in-place. Element lookup (indexing) uses binary search on an internal index data structure and thus takes logarithmic time once the index has been constructed (which is done automatically when needed, or when calling `recordp` on a fresh record value).

Also note that records with duplicate keys are permitted; in such a case the following operations will always operate on the *last* entry for a given key.

type record
: The record type. This is functionally equivalent to `recordp`, but can be used as a type tag on the left-hand side of equations.

recordp x
: Check for record values.

record x
: Normalizes a record. This removes duplicate keys and orders the record by keys (using an apparently random but well-defined order of the key values), so that normalized records are syntactically equal (`===`) if and only if they contain the same hash pairs. For convenience, this function can also be used directly on lists and tuples of hash pairs to convert them to a normalized record value.

x
: The size of a record (number of entries it contains). Duplicate entries are counted. (This is in fact just the standard matrix size operation.)

member x y
: Check whether `x` contains the key `y`.

x **!** y
: Retrieves the (last) value associated with the key `y` in `x`, if any, otherwise throws an `out_of_bound` exception.

x **!!** ys
: Slicing also works as expected, by virtue of the generic definition of slicing provided by the matrix data structure.

insert x (y=>z)
update x y z
: Associate the key `y` with the value `z` in `x`. If `x` already contains the key `y` then the corresponding value is updated (the last such value if `x` contains more than one association for `y`), otherwise a new member is inserted at the end of the record.

delete x y
: Delete the key `y` (and its associated value) from `x`. If `x` contains more than one entry for `y` then the last such entry is removed.

keys x
vals x
: List the keys and associated values of `x`. If the record contains duplicate keys, they are all listed in the order in which they are stored in the record.

Here are a few basic examples:

```
> let r = {x=>5, y=>12};
> r!y; r!![y,x];              // indexing and slicing
12
{12,5}
```

```
> keys r; vals r;             // keys and values of a record
{x,y}
{5,12}
> insert r (x=>99);           // update an existing entry
{x=>99,y=>12}
> insert ans (z=>77);         // add a new entry
{x=>99,y=>12,z=>77}
> delete ans z;               // delete an existing entry
{x=>99,y=>12}
> let r = {r,x=>7,z=>3}; r;   // duplicate key x
{x=>5,y=>12,x=>7,z=>3}
> r!x, r!z;                   // indexing returns the last value of x
7,3
> delete r x;                 // delete removes the last entry for x
{x=>5,y=>12,z=>3}
> record r;                   // normalize (remove dups and sort)
{x=>7,y=>12,z=>3}
> record [x=>5, x=>7, y=>12]; // construct a normalized record from a list
{x=>7,y=>12}
> record (x=>5, x=>7, y=>12); // ... or a tuple
{x=>7,y=>12}
```

More examples can be found in the *Record Data* section in the Pure Manual.

2.1.11 Primitives

This prelude module is a collection of various lowlevel operations, which are implemented either directly by machine instructions or by C functions provided in the runtime. In particular, this module defines the basic arithmetic and logic operations on machine integers, bigints and floating point numbers, as well as various type checking predicates and conversions between different types. Some basic pointer operations are also provided, as well as "sentries" (Pure's flavour of object finalizers) and "references" (mutable expression pointers).

Special Constants

constant `inf`
constant `nan`
: IEEE floating point infinities and NaNs. You can test for these using the `infp` and `nanp` predicates, see Predicates below.

constant `NULL` = pointer 0
: Generic null pointer. (This is actually a built-in constant.) You can also check for null pointers with the `null` predicate, see Predicates.

Arithmetic

The basic arithmetic and logic operations provided by this module are summarized in the following table:

Kind	Operator	Meaning
Arithmetic	`+ -`	addition, subtraction (also unary minus)
	`* /`	multiplication, division (inexact)
	`div mod`	exact int/bigint division/modulus
	`^`	exponentiation (inexact)
Comparisons	`== ~=`	equality, inequality
	`< >`	less than, greater than
	`<= >=`	less than or equal, greater than or equal
Logic	`~`	logical not
	`&& ||`	and, or (short-circuit)
Bitwise	`not`	bitwise not
	`and or`	and, or
	`<< >>`	bit shifts

Precedence and and associativity of the operators can be found in the *operators* table at the beginning of this section.

The names of some operations are at odds with C. Note, in particular, that logical negation is denoted `~` instead of `!` (and, consequently, `~=` denotes inequality, rather than `!=`), and the bitwise operations are named differently. This is necessary because Pure uses `!`, `&` and `|` for other purposes. Also, `/` always denotes inexact (double) division in Pure, whereas the integer division operators are called `div` and `mod`. (`%`, which is not defined by this module, also has a different meaning in Pure; it's the exact division operator, see Rational Numbers.)

The above operations are implemented for int, bigint and, where appropriate, double operands. (Pointer arithmetic and comparisons are provided in a separate module, see Pointer Arithmetic.) The math module (see Mathematical Functions) also provides implementations of the arithmetic and comparison operators for rational, complex and complex rational numbers.

Note that the logical operations are actually implemented as special forms in order to provide for short-circuit evaluation. This needs special support from the compiler to work. The primitives module still provides definitions for these, as well as other special forms like `quote` and the thunking operator `&` so that they may be used as function values and in partial applications, but when used in this manner they lose all their special call-by-name properties; see *Special Forms* in the Pure Manual for details.

A detailed listing of the basic arithmetic and logical operations follows below.

x **+** y
x **-** y
x ***** y
x **/** y
x **^** y
: Addition, subtraction, multiplication, division and exponentiation. The latter two are

inexact and will yield double results.

- x
: Unary minus. This has the same precedence as binary '-' above.

x **div** y
x **mod** y
: Exact int and bigint division and modulus.

x **==** y
x **~=** y
: Equality and inequality.

x **<=** y
x **>=** y
x **>** y
x **<** y
: Comparisons.

~ x
x **&&** y
x **||** y
: Logical negation, conjunction and disjunction. These work with machine ints only and are evaluated in short-circuit mode.

not x
x **and** y
x **or** y
: Bitwise negation, conjunction and disjunction. These work with both machine ints and bigints.

x **<<** k
x **>>** k
: Arithmetic bit shifts. The left operand `x` may be a machine int or a bigint. The right operand `k` must be a machine int and denotes the (nonnegative) number of bits to shift.

Note: This operation may expand to a single machine instruction in the right circumstances, thus the condition that `k` be nonnegative isn't always checked. This may lead to surprising results if you do specify a negative value for `k`. However, in the current implementation bigint shifts do check the sign of `k` and handle it in the appropriate way, by turning a left shift into a corresponding right shift and vice versa.

In addition, the following arithmetic and numeric functions are provided:

abs x
sgn x
: Absolute value and sign of a number.

min x y

max x y
: Minimum and maximum of two values. This works with any kind of values which have the ordering relations defined on them.

succ x
pred x
: Successor (`+1`) and predecessor (`-1`) functions.

gcd x y
lcd x y
: The greatest common divisor and least common multiple functions from the GMP library. These return a bigint if at least one of the arguments is a bigint, a machine int otherwise.

pow x y
: Computes exact powers of ints and bigints. The result is always a bigint. Note that `y` must always be nonnegative here, but see the math module (Mathematical Functions) which deals with the case `y<0` using rational numbers.

Conversions

These operations convert between various types of Pure values.

hash x
: Compute a 32 bit hash code of a Pure expression.

bool x
: Convert a machine integer to a normalized truth value (`0` or `1`).

int x
bigint x
double x
: Conversions between the different numeric types.

pointer x
: Convert a string, int or bigint to a pointer value. Converting a string returns a pointer to the underlying UTF8-encoded C string so that it can be passed to the appropriate C functions. Converting an integer gives a pointer with the given numeric address. This may be used to construct special pointer values such as the null pointer (`pointer 0`).

ubyte x
ushort x
uint x
uint64 x
ulong x
: Convert signed (8/16/32/64) bit integers to the corresponding unsigned quantities. These functions behave as if the value was "cast" to the corresponding unsigned C type, and are most useful for dealing with unsigned integers returned by external C routines. The routines always use the smallest Pure int type capable of holding the result: `int` for `ubyte` and `ushort`, `bigint` for `uint`, `uint64` and `ulong`. All routines take

int parameters. In the case of `uint64`, a bigint parameter is also permitted (which is what the C interface returns for 64 bit values). Also note that `ulong` reduces to either `uint` or `uint64`, depending on the size of `long` for the host architecture.

The following rounding functions work with all kinds of numbers:

`floor` x
`ceil` x
: Floor and ceil.

`round` x
`trunc` x
: Round or truncate to an integer.

`frac` x
: Fractional part (`x-trunc x`).

Note that all these functions return double values for double arguments, so if you need an integer result then you'll have to apply a suitable conversion, as in `int (floor x)`.

Predicates

A syntactic equality test is provided, as well as various type checking predicates. Note that type definitions are provided for most of the type checking predicates which don't denote built-in types; see Prelude Types for details.

`same` x y
x **`===`** y
x **`~==`** y
: Syntactic equality. In contrast to `==` and `~=`, this is defined on all Pure expressions. Basically, two expressions are syntactically equal if they print out the same in the interpreter. In the special case of pointer objects and closures, which do not always have a syntactic representation in Pure, `x` and `y` must be the same object (same pointer value or function).

`typep` ty x
: Generic type checking predicate. This checks whether `x` is of type `ty`, where `ty` is a symbol denoting any of the built-in types (`int`, `bigint` etc.) or any type defined in a `type` definition. (Note that you may have to quote `ty` if it happens to be defined as a variable or parameterless function.)

`intp` x
`bigintp` x
`doublep` x
`stringp` x
`pointerp` x
`matrixp` x
: Predicates to check for the built-in types.

boolp x
: Predicate to check for normalized truth values (`0` and `1`).

charp x
: Predicate to check for single character strings.

numberp x
complexp x
realp x
rationalp x
integerp x
: Additional number predicates. Note some further "semantic" number predicates are defined in the `math` module, see Semantic Number Predicates and Types.

exactp x
inexactp x
: Check whether a number is exact (i.e., doesn't contain any double components).

infp x
nanp x
: Check for `inf` and `nan` values.

null p
: Check for null pointers.

applp x
listp x
rlistp x
tuplep x
: Predicates to check for function applications, lists, proper lists and tuples. Note that `listp` only checks for a toplevel list constructor, whereas `rlistp` also recursively checks the tails of the list; the latter may need time proportional to the list size. The `applp` and `tuplep` predicates look for an application or tuple constructor at the toplevel only, which can always be done in constant time.

funp x
lambdap x
thunkp x
closurep x
: Predicates to check for various kinds of function objects (named, anonymous or thunk). `closurep` checks for any kind of "normal" closure (i.e., named functions and lambdas, but not thunks).

functionp x
: Convenience function to check for "callable" functions. This includes any kind of closure with a nonzero argument count as well as partial (unsaturated) applications of these.

symbolp x
varp x
: Predicates to check for any kind of symbol (this also includes operator and nonfix

symbols) and for free variable symbols, respectively. Note that varp returns true for any symbol which is not an operator or nonfix symbol (i.e., for any symbol that could in principle be bound to a value, either globally or locally). This holds even if the symbol is currently bound to a function, macro or constant.

Inspection

The following operations let you peek at various internal information that the interpreter provides to Pure programs either for convenience or for metaprogramming purposes. They are complemented by the evaluation primitives discussed below, see Eval and Friends.

ans

Retrieve the most recently printed result of a toplevel expression evaluated in the read-eval-print loop. This is just a convenience for interactive usage. Note that the `ans` value will stick around until a new expression is computed. (It is possible to clear the `ans` value with the interactive command `clear ans`, however.) Example:

```
> 1/3;
0.333333333333333
> ans/2;
0.166666666666667
```

__func__

Returns the (lexically) innermost function at the point of the call. This can be either a global function, a local (named) function introduced in a `with` clause or an anonymous function (a lambda). Fails (returning just the literal symbol `__func__` by default) if there is no such function (i.e., if the call is at the toplevel). Note that in contrast to the C99 variable of the same name, this really returns the function value itself in Pure; the `str` function can be used if you need the print name of the function. Examples:

```
> foo x = if x>0 then x else throw __func__;
> foo (-99);
<stdin>, line 2: unhandled exception 'foo' while evaluating 'foo (-99)'
> (\x->x+": "+str __func__) "test";
"test: #<closure 0x7f4a2411db30>"
```

If you want, you can add a default rule for `__func__` which specifies the behaviour when `__func__` gets called at the global level. E.g.:

```
> __func__ = throw "__func__ called at global level";
> __func__;
<stdin>, line 5: unhandled exception '"__func__ called at global level"' while
evaluating '__func__'
```

macro __namespace__

Returns the current namespace at the point of the call. This is implemented as a built-in macro which expands to a string. The empty string is returned in the default namespace. Example:

```
> namespace foo;
> foo = __namespace__;
> namespace;
> show foo::foo
foo::foo = "foo";
> foo::foo;
"foo"
```

macro __list__

This expands a (literal) tuple to a list, preserving embedded tuples in the same way that list values are parsed in the Pure language, cf. *Primary Expressions*. This is provided for the benefit of custom aggregate notations (usually implemented as outfix operators) which are supposed to be parsed like the built-in list and matrix brackets. Example:

```
> outfix (: :);
> def (:x:) = __list__ x;
> (:(1,2),(3,4):);
[(1,2),(3,4)]
```

Note that this macro uses internal information from the parser not available to Pure programs. Thus there's no way to actually define this macro in Pure, which is why it is provided as a builtin instead.

Another rather obscure point that deserves mentioning here is that the special processing of parenthesized expressions happens also if the macro is applied in prefix form. This should rarely be a problem in practice, but if it is then you can use `$` to pass arguments without adding an (undesired) extra level of parentheses:

```
> ((::)) ((1,2),(3,4));
[(1,2,3,4)]
> ((::)) $ (1,2),(3,4);
[(1,2),(3,4)]
```

Note that the first expression is really equivalent to `(:((1,2),(3,4)):)`, *not* `(:(1,2),(3,4):)` which can be specified in prefix form using `$` as shown in the second expression. (Remember that `$` is also implemented as a macro and so is substituted away at macro expansion time in the example above.) The same trick works if for some reason you want to apply `__list__` in a direct fashion:

```
> __list__ ((1,2),(3,4));
[(1,2,3,4)]
> __list__ $ (1,2),(3,4);
[(1,2),(3,4)]
```

macro __locals__

Built-in macro which expands to a list with the local function bindings (`with` clauses) visible at this point in the program. The return value is a list of hash pairs `x=>f` where `x` is the global symbol denoting the function (the symbol is always quoted) and `f` is the function value itself. Example:

```
> __locals__ with foo x = x+1; x = a+b end;
[x=>a+b,foo=>foo]
> f 99 when _=>f = ans!1 end;
100
```

The `__locals__` function is useful for debugging purposes, as well as to implement dynamic environments. It is also used internally to implement the `reduce` macro, see Eval and Friends.

Note that `__locals__` always evaluates parameterless functions and returns the resulting value instead of a closure (as can be seen in the binding `x=>a+b` in the example above). Normally this is what you want, but it can be a problem with parameterless functions involving side effects. In such a case, if you want to evaluate the function at a later time, you'll either have to use a thunk or massage the local function so that it takes a dummy argument such as `()`.

Also note that `__locals__` will use as keys in the resulting list whatever global symbols are in scope at the point of the call. By default, i.e., if no global symbol with the same print name as the local is visible at the point of the call, a symbol in the default namespace is used, as we've seen above. Otherwise the result may be also be a qualified symbol if such a symbol has already been declared or defined at the point of the call. For instance:

```
> namespace foo;
> public foo;
> __locals__ with foo x = x+1 end;
[foo::foo=>foo]
```

This behaviour may be a bit surprising at first sight, but is consistent with the way the interpreter performs its symbol lookup, see *Symbol Lookup and Creation* for details.

The following functions allow you to inspect or modify the function, type, macro, constant and variable definitions of the running program. This uses a special meta representation for rewriting rules and definitions. Please see the *Macros* section in the Pure manual for details. Also note that these operations are subject to some limitations, please check the remarks concerning `eval` and `evalcmd` in the following subsection for details.

get_fundef sym
get_typedef sym
get_macdef sym
: If the given symbol is defined as a function, type or macro, return the corresponding list of rewriting rules. Otherwise return the empty list.

get_interface sym
get_interface_typedef sym
: If the given symbol is defined as an interface type, return its definition; otherwise return the empty list. `get_interface` returns the list of patterns used to declare the type, while `get_interface_typedef` returns the actual list of type rules, in the same format as with `get_typedef`. Note that the latter may be empty even if the type is defined, meaning that the type hasn't been instantiated yet, see *Interface Types* for details. Also

note that Pure allows you to have *both* an interface and a regular (concrete) definition of a type, in which case `get_typedef` and `get_interface_typedef` may both return nonempty (and usually different) results.

get_vardef sym
get_constdef sym
: If the given symbol is defined as a variable or constant, return the corresponding definition as a singleton list of the form `[sym --> value]`. Otherwise return the empty list.

The following functions may fail in case of error, in which case `lasterr` is set accordingly (see Eval and Friends below).

add_fundef rules
add_typedef rules
add_macdef rules
: Add the given rewriting rules (given in the same format as returned by the `get_fundef`, `get_typedef` and `get_macdef` functions above) to the running program.

add_fundef_at r rules
add_typedef_at r rules
add_macdef_at r rules
: Same as above, but add the given rewriting rules at (i.e., before) the given rule `r` (which must already exist, otherwise the call fails). Note that all added rules must have the same head symbol on the left-hand side, which matches the head symbol on the left-hand side of `r`.

add_interface sym patterns
: Add the given patterns to the interface type `sym` (given as a symbol). If the interface type doesn't exist yet, it will be created.

add_interface_at sym p patterns
: Same as above, but add the given patterns at (i.e., before) the given pattern `p` (the given interface type must already exist and contain the given pattern, otherwise the call fails).

add_vardef rules
add_constdef rules
: Define variables and constants. Each rule must take the form `sym --> value` with a symbol on the left-hand side (no pattern matching is performed by these functions).

The following functions may be used to delete individual rewriting rules, interface type patterns or variable and constant symbols.

del_fundef rule
del_typedef rule
del_macdef rule
: Delete the given rewriting rule (given in the same format as returned by the `get_fundef`, `get_typedef` and `get_macdef` functions) from the running program. Returns `()` if successful, fails otherwise.

del_interface sym pattern
: Delete the given pattern from the given interface type. Returns `()` if successful, fails

otherwise.

del_vardef sym
del_constdef sym
: Delete variables and constants, given by their (quoted) symbols. Returns `()` if successful, or fails if the symbol isn't defined (or defined as a different kind of symbol).

The prelude also provides some functions to retrieve various attributes of a function symbol which determine how the operation is applied to its operands or arguments. These functions all take a single argument, the symbol or function object to be inspected, and return an integer value.

nargs x
: Get the argument count of a function object, i.e., the number of arguments it expects. Returns 0 for thunks and saturated applications, -1 for over-saturated applications and non-functions.

arity x
: Determine the arity of an operator symbol. The returned value is 0, 1 or 2 for nullary, unary and binary symbols, respectively, -1 for symbols without a fixity declaration or other kinds of objects.

fixity f
: Determine the fixity of an operator symbol. The fixity is encoded as an integer `10*n+m` where `n` is the precedence level (ranging from `0` to `PREC_MAX`, where `PREC_MAX` denotes the precedence of primary expressions, 16777216 in the current implementation) and `m` indicates the actual fixity at each level, in the order of increasing precedence (0 = infix, 1 = infixl, 2 = infixr, 3 = prefix, 4 = postfix). The fixity value of nonfix and outfix symbols, as well as symbols without a fixity declaration, is always given as `10*PREC_MAX`, and the same value is also reported for non-symbol objects. Infix, prefix and postfix symbols always have a `fixity` value less than `10*PREC_MAX`. (`PREC_MAX` isn't actually defined as a constant anywhere, but you can easily do that yourself by setting `PREC_MAX` to the fixity value of any nonfix symbol or non-symbol value, e.g.: `const PREC_MAX = fixity [];`)

Note that only closures (i.e., named and anonymous functions and thunks) have a defined argument count in Pure, otherwise `nargs` returns -1 indicating an unknown argument count. Partial applications of closures return the number of remaining arguments, which may be zero to indicate a **saturated** (but unevaluated) application, or -1 for **over-saturated** and constructor applications. (Note that in Pure a saturated application may also remain unevaluated because there is no definition for the given combination of arguments and thus the expression is in normal form, or because the application was quoted. If such a normal form application is then applied to some "extra" arguments it becomes over-saturated.)

The value returned by `nargs` always denotes the actual argument count of the given function, regardless of the declared arity if the function also happens to be an operator symbol. Often these will coincide (as, e.g., in the case of `+` which is a binary operator and also expects two arguments). But this is not necessarily the case, as shown in the following example of a binary operator which actually takes *three* arguments:

```
> infix 0 oops;
> (oops) x y z = x*z+y;
> arity (oops);
2
> nargs (oops);
3
> nargs (5 oops 8);
1
> map (5 oops 8) (1..5);
[13,18,23,28,33]
```

Eval and Friends

Pure provides some rather powerful operations to convert between Pure expressions and their string representation, and to evaluate quoted expressions (`'x`). The string conversions `str`, `val` and `eval` also provide a convenient means to serialize Pure expressions, e.g., when terms are to be transferred to/from persistent storage. (Note, however, that this has its limitations. Specifically, some objects like pointers and anonymous functions do not have a parsable string representation. Also see the Expression Serialization section for some dedicated serialization operations which provide a more compact binary serialization format.)

`str` x
: Yields the print representation of an expression in Pure syntax, as a string.

`val` s
: Parses a single simple expression, specified as a string in Pure syntax, and returns the result as is, without evaluating it. Note that this is much more limited than the `eval` operation below, as the expression must not contain any of the special constructs (conditional expressions, `when`, `with`, etc.), unless they are quoted.

`eval` x
: Parses any expression, specified as a string in Pure syntax, and returns its value. In fact, `eval` can also parse and execute arbitrary Pure code. In that case it will return the last computed expression, if any. Alternatively, `eval` can also be invoked on a (quoted) Pure expression, which is recompiled and then evaluated. Exceptions during evaluation are reported back to the caller.

Note: The use of `eval` and `evalcmd` (as well as `add_fundef`, `add_typedef` etc. from the preceding subsection) to modify a running program breaks referential transparency and hence these functions should be used with care. Also, none of the inspection and mutation capabilities provided by these operations will work in batch-compiled programs, please check the *Batch Compilation* section in the Pure manual for details. Moreover, using these operations to modify or delete a function which is currently being executed results in undefined behaviour.

`evalcmd` x

Like `eval`, but allows execution of interactive commands and returns their captured output as a string. No other results are returned, so this operation is most useful for executing Pure definitions and interactive commands for their side-effects. (At this time, only the regular output of a few commands can be captured, most notably `bt`, `clear`, `mem`, `save` and `show`; otherwise the result string will be empty.)

`lasterr`
Reports errors in `val`, `eval` and `evalcmd` (as well as in `add_fundef` et al, described in the previous subsection). This string value will be nonempty iff a compilation or execution error was encountered during the most recent invocation of these functions. In that case each reported error message is terminated with a newline character.

`lasterrpos`
Gives more detailed error information. This returns a list of the individual error messages in `lasterr`, along with the position of each error (if available). Each list item is either just a string (the error message, with any trailing newline stripped off) if no error position is available, or a tuple of the form `msg,file,l1,c1,l2,c2` where `msg` is the error message, `file` the name of the file containing the error (which will usually be `"<stdin>"` indicating that the error is in the source string, but may also be a proper filename of a module imported in the evaluated code), `l1,c1` denotes the beginning of the range with the errorneous construct (given as line and column indices) and `l2,c2` its end (or rather the character position following it). For convenience, both line and column indices are zero-based, in order to facilitate extraction of the text from the actual source string.

Note: The indicated error positions are only approximate, and may in many cases span an entire syntactic construct (such as a subexpression or even an entire function definition) containing the error. Also, the end of the range may sometimes point one token past the actual end of the construct. (These limitations are due to technical restrictions in the parser; don't expect them to go away anytime soon.)

Examples:

```
> str (1/3);
"0.333333333333333"
> val "1/3";
1/3
> eval "1/3";
0.333333333333333
> eval ('(1/3));
0.333333333333333
> evalcmd "show evalcmd";
"extern expr* evalcmd(expr*);\n"
> eval "1/3)";
eval "1/3)"
> lasterr;
"<stdin>, line 1: syntax error, unexpected ')', expecting '=' or '|'\n"
> lasterrpos;
[("<stdin>, line 1: syntax error, unexpected ')', expecting '=' or '|'",
```

```
"<stdin>",0,3,0,4)]
```

In addition to `str`, the prelude also provides the following function for pretty-printing the internal representation used to denote quoted specials. This is commonly used in conjunction with the `__show__` function, please see the *Macros* section in the Pure manual for details.

__str__ x
: Pretty-prints special expressions.

Example:

```
> __str__ ('__lambda__ [x __type__ int] (x+1));
"\\x::int -> x+1"
```

The `evalcmd` function is commonly used to invoke the `show` and `clear` commands for metaprogramming purposes. The prelude provides the following two convenience functions to make this easy:

globsym pat level
: This uses `evalcmd` with the `show` command to list all defined symbols matching the given glob pattern. A definition level may be specified to restrict the context in which the symbol is defined; a level of 0 indicates that all symbols are eligible (see the description of the `show` command in the Pure manual for details). The result is the list of all matching (quoted) symbols.

clearsym sym level
: This uses `evalcmd` with the `clear` command to delete the definition of the given symbol at the given definition level. No glob patterns are permitted here. The `sym` argument may either be a string or a literal (quoted) symbol.

Example:

```
> let x,y = 77,99;
> let syms = globsym "[a-z]" 0; syms;
[x,y]
> map eval syms;
[77,99]
> do (flip clearsym 0) syms;
()
> globsym "[a-z]" 0;
[]
> x,y;
x,y
```

The following functions are useful for doing symbolic expression simplification.

macro reduce x
: Reevaluates an expression in a local environment. This dynamically rebinds function symbols in the given expression to whatever local function definitions are in effect at the point of the `reduce` call. Note that `reduce` is actually implemented as a macro

which expands to the `reduce_with` primitive (see below), using the `__locals__` builtin to enumerate the bindings which are in effect at the call site.

`reduce_with` env x
: Like `reduce` above, but takes a list of replacements (given as hash pairs `u=>v`) as the first argument. The `reduce` macro expands to `reduce_with __locals__`.

The `reduce` macro provides a restricted form of dynamic binding which is useful to implement local rewriting rules. It is invoked without parameters and expands to the curried call `reduce_with __locals__` of the `reduce_with` primitive, which takes one additional argument, the expression to be rewritten. The following example shows how to expand or factorize an expression using local rules for the laws of distributivity:

```
expand = reduce with
  (a+b)*c = a*c+b*c;
  a*(b+c) = a*b+a*c;
end;

factor = reduce with
  a*c+b*c = (a+b)*c;
  a*b+a*c = a*(b+c);
end;

expand ((a+b)*2); // yields a*2+b*2
factor (a*2+b*2); // yields (a+b)*2
```

Note that instances of locally bound functions are substituted back in the computed result, thus the instances of `*` and `+` in the results `a*2+b*2` and `(a+b)*2` shown above denote the corresponding globals, not the local incarnations of `*` and `+` defined in `expand` and `factor`, respectively.

`reduce` also adjusts to quoted arguments. In this case, the local rules are applied as usual, but back-substituted globals are *not* evaluated in the result:

```
> expand ((a+1)*2);
a*2+2
> expand ('((a+1)*2));
a*2+1*2
```

Note that `reduce` only takes into account local *function* bindings from `with` clauses, local *variable* bindings do not affect its operation in any way:

```
> let y = [x,x^2,x^3];
> reduce y when x = u+v end;
[x,x^2,x^3]
```

However, in such cases you can perform the desired substitution by turning the `when` into a `with` clause:

```
> reduce y with x = u+v end;
[u+v,(u+v)^2,(u+v)^3]
```

Or you can just invoke the underlying `reduce_with` builtin directly, with the desired substitutions given as hash pairs in the first argument:

```
> reduce_with [x=>u+v] y;
[u+v,(u+v)^2,(u+v)^3]
```

Expression Serialization

Like `str` and `eval`, the following `blob` and `val` operations can be used to safely transfer expression data to/from persistent storage and between different processes (using, e.g., POSIX shared memory, pipes or sockets). However, `blob` and `val` use a binary format which is usually much more compact and gets processed much faster than the string representations used by `str` and `eval`. Also, `val` offers some additional protection against transmission errors through a crc check. (The advantage of the string representation, however, is that it's readable plain text in Pure syntax.)

`blob` x
: Stores the contents of the given expression as a binary object. The return value is a cooked pointer which frees itself when garbage-collected.

`val` p
: Reconstructs a serialized expression from the result of a previous invocation of the `blob` function.

`blobp` p
: Checks for a valid `blob` object. (Note that `val` may fail even if `blobp` returns `true`, because for performance reasons `blobp` only does a quick plausibility check on the header information of the blob, whereas `val` also performs a crc check and verifies data integrity.)

`#` p
`blob_size` p
`blob_crc` p
: Determines the size (in bytes) and crc checksum of a blob, respectively. `blob_size` always returns a bigint, `blob_crc` a machine int (use `uint` on the latter to get a proper unsigned 32 bit value). For convenience, `#p` is defined as an alias for `blob_size p` on `blob` pointers.

Example:

```
> let b = blob {"Hello, world!", 1/3, 4711, NULL};
> b; #b; uint $ blob_crc b;
#<pointer 0x141dca0>
148L
3249898239L
> val b;
{"Hello, world!",0.333333333333333,4711,#<pointer 0>}
```

Please note that the current implementation has some limitations:

- Just as with `str` and `eval`, runtime data (local closures and pointers other than the `NULL` pointer) can't be serialized, causing `blob` to fail. However, it *is* possible to transfer a global function, provided that the function exists (and is the same) in both the sending and the receiving process. (This condition can't be verified by `val` and thus is at the programmer's responsibilty.)
- Sharing of subexpressions will in general be preserved, but sharing of list and tuple *tails* will be lost (unless the entire list or tuple is shared).
- The `val` function may fail to reconstruct the serialized expression even for valid blobs, if there is a conflict in symbol fixities between the symbol tables of the sending and the receiving process. To avoid this, make sure that symbol declarations in the sending and the receiving script match up.

Other Special Primitives

`exit` status
: Terminate the program with the given status code.

`throw` x
: Throw an exception, cf. *Exception Handling*.

`break`
`trace`
: Trigger the debugger from a Pure program, cf. *Debugging*. Note that these routines only have an effect if the interpreter is run in debugging mode, otherwise they are no-ops. The debugger will be invoked at the next opportunity (usually when a function is called or a reduction is completed).

`force` x
: Force a thunk (x&), cf. *Special Forms*. This usually happens automagically when the value of a thunk is needed.

Pointer Operations

The prelude provides a few basic operations on pointers which make it easy to interface to external C functions. For more advanced uses, the library also includes the `pointers` module which can be imported explicitly if needed, see Pointer Arithmetic below.

`addr` symbol
: Get the address of a C symbol (given as a string) at runtime. The library containing the symbol must already be loaded. Note that this can in fact be any kind of externally visible C symbol, so it's also possible to get the addresses of global variables. The result is returned as a pointer. The function fails if the symbol was not found.

`calloc` nmembers size
`malloc` size
`realloc` ptr size

free ptr
: Interface to `malloc`, `free` and friends. These let you allocate dynamic buffers (represented as Pure pointer values) for various purposes.

The following functions perform direct memory accesses through pointers. Their primary use is to interface to certain C library functions which take or return data through pointers. It goes without saying that these operations should be used with utmost care. No checking is done on the pointer types, so it is the programmer's responsibility to ensure that the pointers actually refer to the corresponding type of data.

get_byte ptr
get_short ptr
get_int ptr
get_int64 ptr
get_long ptr
get_float ptr
get_double ptr
get_string ptr
get_pointer ptr
: Return the integer, floating point, string or generic pointer value at the memory location indicated by `ptr`.

put_byte ptr x
put_short ptr x
put_int ptr x
put_int64 ptr x
put_long ptr x
put_float ptr x
put_double ptr x
put_string ptr x
put_pointer ptr x
: Change the integer, floating point, string or generic pointer value at the memory location indicated by `ptr` to the given value `x`.

Sentries

Sentries are Pure's flavour of object **finalizers**. A sentry is simply an object (usually a function) which gets applied to the target expression when it is garbage-collected. This is useful to perform automatic cleanup actions on objects with internal state, such as files. Pure's sentries are *much* more useful than finalizers in other garbage-collected languages, since it is guaranteed that they are called as soon as an object "goes out of scope", i.e., becomes inaccessible.

sentry f x
: Places a sentry `f` at an expression `x` and returns the modified expression.

clear_sentry x
: Removes the sentry from an expression `x`.

get_sentry x
: Returns the sentry of an expression x (if any, fails otherwise).

As of Pure 0.45, sentries can be placed on any Pure expression. The sentry itself can also be any type of object (but usually it's a function). Example:

```
> using system;
> sentry (\_->puts "I'm done for!") (1..3);
[1,2,3]
> clear ans
I'm done for!
```

Note that setting a finalizer on a global symbol won't usually be of much use since such values are cached by the interpreter. (However, the sentry *will* be invoked if the symbol gets recompiled because its definition has changed. This may be useful for some purposes.)

In Pure parlance, we call an expression **cooked** if a sentry has been attached to it. The following predicate can be used to check for this condition. Also, there is a convenience function to create cooked pointers which take care of freeing themselves when they are no longer needed.

cookedp x
: Check whether a given object has a sentry set on it.

cooked ptr
: Create a pointer which disposes itself after use. This is just a shorthand for `sentry free`. The given pointer `ptr` must be `malloc`ed to make this work.

Example:

```
> using system;
> let p = cooked (malloc 1024);
> cookedp p;
1
> get_sentry p;
free
> clear p
```

Besides their use as finalizers, sentries can also be handy in other circumstances, when you need to associate an expression with another, "invisible" value. In this case the sentry is usually some kind of data structure instead of a function to be executed at finalization time. For instance, here's how we can employ sentries to implement hashing of function values:

```
using dict;
hashed f x = case get_sentry f of
               h::hdict = h!x if member h x;
               _ = y when y = f x; sentry (update h x y) f
                       when h = case get_sentry f of
                                  h::hdict = h; _ = emptyhdict
                                end;
                       end;
                     end;
             end;
```

E.g., consider the naive recursive definition of the Fibonacci function:

```
fib n::int = if n<=1 then 1 else fib (n-1)+fib (n-2);
```

A hashed version of the Fibonacci function can be defined as follows:

```
let hfib = hashed f with
  f n::int = if n<=1 then 1 else hfib (n-1)+hfib (n-2)
end;
```

This turns the naive definition of the Fibonacci function (which has exponential time complexity) into a linear time operation:

```
> stats
> fib 35;
14930352
4.53s
> hfib 35;
14930352
0.25s
```

Finally, note that there can be only one sentry per expression but, building on the operations provided here, it's easy to design a scheme where sentries are chained. For instance:

```
chain_sentry f x = sentry (h (get_sentry x)) x with
  h g x = g x $$ f x;
end;
```

This invokes the original sentry before the chained one:

```
> using system;
> f _ = puts "sentry#1"; g _ = puts "sentry#2";
> let p = chain_sentry g $ sentry f $ malloc 10;
> clear p
sentry#1
sentry#2
```

You can chain any number of sentries that way. This scheme should work in most cases in which sentries are used just as finalizers. However, there are other uses, like the "hashed function" example above, where you'd like the original sentry to stay intact. This can be achieved by placing the new sentry as a sentry on the *original sentry* rather than the expression itself:

```
attach_sentry f x = sentry (sentry f (get_sentry x)) x;
```

This requires that the sentry will actually be garbage-collected when its hosting expression gets freed, so it will *not* work if the original sentry is a global:

```
> let p = attach_sentry g $ sentry f $ malloc 10;
> clear p
sentry#1
```

However, the attached sentry will work ok if you can ensure that the original sentry is a (partial or constructor) application. E.g.:

```
> let p = attach_sentry g $ sentry (f$) $ malloc 10;
> clear p
sentry#1
sentry#2
```

Tagged Pointers

As of Pure 0.45, the C interface now fully checks pointer parameter types at runtime (see the *C Types* section in the Pure Manual for details). To these ends, pointer values are internally tagged to keep track of the pointer types. The operations described in this section give you access to these tags in Pure programs. At the lowest level, a pointer tag is simply a machine int associated with a pointer value. The default tag is 0, which denotes a generic pointer value, i.e., `void*` in C. The following operations are provided to create such tags, and set, get or verify the tag of a pointer value.

ptrtag t x
: Places an integer tag `t` at an expression `x` and returns the modified expression. `x` must be a pointer value.

get_ptrtag x
: Retrieves the tag associated with `x`.

check_ptrtag t x
: Compares the tag associated with `x` against `t` and returns true iff the tags match. If `x` is a pointer value, this is equivalent to `get_ptrtag x==t || null x && get_ptrtag x==0`.

make_ptrtag
: Returns a new, unique tag each time it is invoked.

Examples:

```
> let p = malloc 10;
> get_ptrtag p; // zero by default
0
> let t = make_ptrtag; t;
12
> ptrtag t p;
#<pointer 0xc42da0>
> get_ptrtag p;
12
> check_ptrtag t p;
1
> check_ptrtag 0 p;
0
```

Note that in the case of a non-`NULL` pointer, `check_ptrtag` just tests the tags for equality. On the other hand, a generic `NULL` pointer, like in C, is considered compatible with all pointer

types:

```
> let t1 = make_ptrtag; t1;
13
> check_ptrtag t1 p;
0
> check_ptrtag t1 NULL;
1
> get_ptrtag NULL;
0
```

The operations above are provided so that you can design your own, more elaborate type systems for pointer values if the need arises. However, you'll rarely have to deal with pointer tags at this level yourself. For most applications, it's enough to inspect the type of a Pure pointer and maybe modify it by "casting" it to a new target type. The following high-level operations provide these capabilities.

pointer_tag ty
pointer_tag x
: Returns the pointer tag for the given type `ty`, denoted as a string, or the given pointer value `x`. In the former case, the type should be specified in the C-like syntax used in `extern` declarations; a new tag will be created using `make_ptrtag` if needed. In the latter case, `pointer_tag` simply acts as a frontend for `get_ptrtag` above.

pointer_type tag
pointer_type x
: Returns the type name associated with the given int value `tag` or pointer value `x`. Please note that this may be `NULL` in the case of an "anonymous" tag, which may have been created with `make_ptrtag` above, or if the tag is simply unknown because it hasn't been created yet.

pointer_cast tag x
pointer_cast ty x
: Casts `x` (which must be a pointer value) to the given pointer type, which may be specified either as a tag or a string denoting the type name. This returns a new pointer value with the appropriate type tag on it (the tag on the original pointer value `x` isn't affected by this operation).

Example:

```
> let p = malloc 10;
> let q = pointer_cast "char*" p;
> map pointer_type [p,q];
["void*","char*"]
> map pointer_tag [p,q];
[0,1]
> map pointer_type (0..make_ptrtag-1);
["void*","char*","void**","char**","short*","short**","int*","int**",
"float*","float**","double*","double**"]
```

(The last command shows a quick and dirty way to retrieve the currently defined type tags in the interpreter. This won't work in batch-compiled scripts, however, since in this case the

range of type tags is in general non-contiguous.)

If you have to do many casts to a given type, you can avoid the overhead of repeatedly looking up the type name by assigning the tag to a variable, which can then be passed to `pointer_cast` instead:

```
> let ty = pointer_tag "long*";
> pointer_cast ty p, pointer_cast ty q;
```

Note that you have to be careful when casting a cooked pointer, because `pointer_cast` may have to create a copy of the original pointer value in order not to clobber the original type tag. The sentry will then still be with the original cooked pointer value, thus you have to ensure that this value survives its type-cast duplicate. It's usually best to apply the cast right at the spot where the pointer gets passed to an external function, e.g.:

```
> extern char *gets(char*);
> let p = cooked $ malloc 1000;
> gets (pointer_cast "char*" p);
```

Such usage is always safe. If this approach isn't possible, you might want to use the lowlevel `ptrtag` operation instead. (This will clobber the type tag of the pointer, but you can always change it back afterwards.)

Expression References

Expression references provide a kind of mutable data cells which can hold any Pure expression. If you need these, then you're doomed. ;-) However, they can be useful as a last resort when you need to keep track of some local state or interface to the messy imperative world. Pure's references are actually implemented as expression pointers so that you can readily pass them as pointers to a C function which expects a `pure_expr**` parameter. This may even be useful at times.

type ref
: The type of expression references. This is a subtype of the `pointer` type.

ref x
: Create a reference pointing to `x` initially.

put r x
: Set a new value `x`, and return that value.

get r
: Retrieve the current value `r` points to.

unref r
: Purge the referenced object and turn the reference into a dangling pointer. (This is used as a sentry on reference objects and shouldn't normally be called directly.)

refp x
: Predicate to check for reference values.

Note that manually changing or removing the `unref` sentry of a reference turns the reference into just a normal pointer object and renders it unusable as a reference. Doing this will also leak memory, so don't!

There is another pitfall with expression references, namely that they can be used to create cyclic chains which currently can't be reclaimed by Pure's reference-counting garbage collector. For instance:

```
> using system;
> done r = printf "done %s\n" (str r);
> let x = ref ();
> let y = ref (sentry done 2,x);
> put x (sentry done 1,y);
1,#<pointer 0x3036400>
```

At this point `x` points to `y` and vice versa. If you now purge the `x` and `y` variables then Pure won't be able to reclaim the cycle, resulting in a memory leak (you can verify this by noting that the sentries are not being called). To prevent this, you'll have to break the cycle first:

```
> put y 3;
done 2
3
> clear x y
done 1
```

Note that, in a way, sentries work similar to expression references and thus the same caveats apply there. Having a limited amount of cyclic references won't do any harm. But if they can grow indefinitely then they may cause problems with long-running programs due to memory leakage, so it's a good idea to avoid such cycles if possible.

Pointer Arithmetic

The pointers.pure module provides the usual C-style pointer arithmetic and comparisons of pointer values. This module normally is not included in the prelude, so to use these operations, you have to add the following import declaration to your program:

```
using pointers;
```

The module overloads the comparison and some of the arithmetic operators (cf. Arithmetic) so that they can be used to compare pointers and to perform C-style pointer arithmetic. To these ends, some conversions between pointers and numeric types are also provided.

int p
bigint p
: Convert a pointer to an int or bigint, giving its numeric address value, which usually denotes a byte offset relative to the beginning of the memory of the executing process. This value can then be used in arithmetic operations and converted back to a pointer using the `pointer` function from the prelude. (Note that to make this work on 64 bit systems, you'll have to convert the pointer values to bigints.)

p **+** n
p **-** n
p **-** q
: Pointer arithmetic. `p+n` and `p-n` offsets a pointer `p` by the given integer `n` denoting the amount of bytes. In addition, `p-q` returns the byte offset between two pointers `p` and `q`. Note that, in contrast to C pointer arithmetic which also takes into account the base type of the pointer, the Pure operations always use byte offsets, no matter what type of pointer (as given by the pointer tag) is passed to these operations.

p **==** q
p **~=** q
: Pointer equality and inequality. This is exactly the same as syntactic equality on pointers.

p **<=** q
p **>=** q
p **>** q
p **<** q
: Pointer comparisons. One pointer `p` is considered to be "less" than another pointer `q` if it represents a "lower" address in memory, i.e., if the byte offset `p-q` is negative.

2.2 Mathematical Functions

The math.pure module provides Pure's basic math routines. It also defines complex and rational numbers.

2.2.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using math;
```

2.2.2 Basic Math Functions

The module defines the following real-valued constants:

constant e = 2.71828...
: Euler's number.

constant pi = 3.1415...
: Ludolph's number.

It also provides a reasonably comprehensive (pseudo) random number generator which uses the Mersenne twister to avoid bad generators present in some C libraries.

Please note that as of Pure 0.41, the runtime library includes a newer release of the Mersenne twister which fixes issues with some kinds of seed values, and will yield different values for

given seeds. Also, the `random31` and `random53` functions have been added as a convenience to compute unsigned 31 bit integers and 53 bit double values, and the `srandom` function now also accepts an int matrix as seed value.

`random`
Return 32 bit pseudo random ints in the range `-0x80000000..0x7fffffff`.

`random31`
Return 31 bit pseudo random ints in the range `0..0x7fffffff`.

`random53`
Return pseudo random doubles in the range `[0,1)` with 53 bits resolution.

`srandom` seed
Sets the seed of the generator to the given 32 bit integer. You can also specify longer seeds using a nonempty row vector, e.g.: `srandom {0x123, 0x234, 0x345, 0x456}`.

The following functions work with both double and int/bigint arguments. The result is always a double. For further explanations please see the descriptions of the corresponding functions from the C math library.

`sqrt` x
The square root function.

`exp` x
`ln` x
`log` x
Exponential function, natural and decadic logarithms.

`sin` x
`cos` x
`tan` x
Trigonometric functions.

`asin` x
`acos` x
`atan` x
Inverse trigonometric functions.

`atan2` y x
Computes the arcus tangent of `y/x`, using the signs of the two arguments to determine the quadrant of the result.

`sinh` x
`cosh` x
`tanh` x
Hyperbolic trigonometric functions.

`asinh` x
`acosh` x
`atanh` x
Inverse hyperbolic trigonometric functions.

2.2.3 Complex Numbers

x **+:** y
r **<:** t
: Complex number constructors.

constant i = 0+:1
: Imaginary unit.

We provide both rectangular (`x+:y`) and polar (`r<:a`) representations, where `(x,y)` are the Cartesian coordinates and `(r,t)` the radius (absolute value) and angle (in radians) of a complex number, respectively. The `+:` and `<:` constructors (declared in the prelude) bind weaker than all other arithmetic operators and are non-associative.

The polar representation `r<:t` is normalized so that `r` is always nonnegative and `t` falls in the range `-pi<t<=pi`.

The constant `i` is provided to denote the imaginary unit `0+:1`.

The arithmetic operations `+`, `*` etc. and the equality relations `==` and `~=` work as expected, and the square root, exponential, logarithms, trigonometric and hyperbolic trigonometric functions (see Basic Math Functions) are extended to complex numbers accordingly. These do *not* rely on complex number support in the C library, but should still conform to IEEE 754 and POSIX, provided that the C library provides a standards-compliant implementation of the basic math functions.

The following operations all work with both the rectangular and the polar representation, promoting real (double, int/bigint) inputs to complex where appropriate. When the result of an operation is again a complex number, it generally uses the same representation as the input (except for explicit conversions). Mixed rect/polar and polar/rect arithmetic always returns a rect result, and mixed complex/real and real/complex arithmetic yields a rect or polar result, depending on what the complex input was.

complex x
: Convert any kind of number to a complex value.

polar z
rect z
: Convert between polar and rectangular representations.

cis t
: Create complex values on the unit circle. Note: To quickly compute `exp (x+:y)` in polar form, use `exp x <: y`.

abs z
arg z
: Modulus (absolute value) and argument (angle, a.k.a. phase). Note that you can also find both of these in one go by converting to polar form.

re z
im z
: Real and imaginary part.

conj z
: Complex conjugate.

Examples:

```
> using math;
> let z = 2^(1/i); z;
0.769238901363972+:-0.638961276313635
> let z = ln z/ln 2; z;
0.0+:-1.0
> abs z, arg z;
1.0,-1.5707963267949
> polar z;
1.0<:-1.5707963267949
```

Please note that, as the `+:` and `<:` constructors bind weaker than the other arithmetic operators, complex numbers *must* be parenthesized accordingly, e.g.:

```
> (1+:2)*(3+:4);
-5+:10
```

2.2.4 Rational Numbers

x **%** y
: Exact division operator and rational number constructor.

Pure's rational numbers are constructed with the **exact division** operator `%` (declared in the prelude) which has the same precedence and fixity as the other division operators.

The `%` operator returns a rational or complex rational for any combination of integer, rational and complex integer/rational arguments, provided that the denominator is nonzero (otherwise it behaves like `x div 0`, which will raise an exception). Machine int operands are always promoted to bigints, thus normalized rationals always take the form `x%y` where both the numerator `x` and the denominator `y` are bigints. For other numeric operands `%` works just like `/`. Rational results are normalized so that the sign is always in the numerator and numerator and denominator are relatively prime. In particular, a rational zero is always represented as `0L%1L`.

The usual arithmetic operations and equality/order relations are extended accordingly, as well as the basic math functions and the rounding functions, and will return exact (rational or complex rational) results where appropriate. Rational operations are implemented using the GMP bigint functions where possible, and thus are reasonably fast.

In addition, the module also provides following operations:

rational x
: Converts a real or complex value `x` to a rational or complex rational. Note that the conversion from double values doesn't do any rounding, so it is guaranteed that converting the resulting rational back to a double reconstructs the original value.

Conversely, the `int`, `bigint`, `double`, `complex`, `rect`, `polar` and `cis` conversion functions are overloaded so that they convert a rational to one of the other number types.

num x
den x
Numerator and denominator of a rational x.

Examples:

```
> using math;
> 5%7 + 2%3;
29L%21L
> 3%8 - 1%3;
1L%24L
> pow (11%10) 3;
1331L%1000L
> let x = pow 3 (-3); x;
1L%27L
> num x, den x;
1L,27L
> rational (3/4);
3L%4L
```

Note that doubles can't represent most rationals exactly, so conversion from double to rational *will* yield funny results in many cases (which are still accurate up to rounding errors). For instance:

```
> let x = rational (1/17); x;
4238682002231055L%72057594037927936L
> num x/den x;
0.0588235294117647
> double (1%17);
0.0588235294117647
```

2.2.5 Semantic Number Predicates and Types

In difference to the syntactic predicates in Primitives, these check whether the given value can be represented as an object of the given target type (up to rounding errors). Note that if `x` is of syntactic type `X`, then it is also of semantic type `X`. Moreover, `intvalp x => bigintvalp x => ratvalp x => realvalp x => compvalp x <=> numberp x`.

compvalp x
Check for complex values (this is the same as `numberp`).

realvalp x
Check for real values (`im x==0`).

ratvalp x
Check for rational values (same as `realvalp`, except that IEEE 754 infinities and NaNs are excluded).

bigintvalp x
: Check for "big" integer values which can be represented as a bigint.

intvalp x
: Check for "small" integer values which can be represented as a machine int.

type compval
type realval
type ratval
type bigintval
type intval
: Convenience types for the above predicates. These can be used as type tags on the left-hand side of an equation to match numeric values for which the corresponding predicate yields `true`.

2.3 Enumerated Types

Enumerated types, or **enumerations** for short, are algebraic types consisting only of nullary constructor symbols. The operations of this module equip such types with the necessary function definitions so that the members of the type can be employed in arithmetic operations, comparisons, etc. in the same way as the predefined enumerated types such as integers and characters. This also includes support for arithmetic sequences.

Please note that this module is not included in the prelude by default, so you have to use the following import declaration to get access to its operations:

```
using enum;
```

The following operations are provided:

enum sym
: The given symbol must denote an algebraic type consisting only of nonfix symbols. `enum` adds the necessary rules for making members of the type work with enumerated type operations such as `ord`, `succ`, `pred`, comparisons, basic arithmetic and arithmetic sequences. It also defines `sym` as an ordinary function, called the **enumeration function** of the type, which maps ordinal numbers to the corresponding members of the type (`sym 0` yields the first member of the type, `sym 1` the second, etc.). The members of the type are in the same order as given in the definition of the type.

defenum sym [symbols,...]
: A convenience function which declares a type `sym` with the given elements and invokes `enum` on it to make it enumerable in one go.

enumof sym
: Given a member of an enumerated type as defined with `enum`, this returns the enumeration function of the type. Rules for this function are generated automatically by `enum`.

type enum
: The type of all enumerated type members. This is actually implemented as an interface type. It matches members of all enumerated types constructed with `enum`.

enump x
: Predicate to check for enumerated type members.

For instance, consider:

```
nonfix sun mon tue wed thu fri sat;
type day sun | day mon | day tue | day wed | day thu | day fri | day sat;
```

Once the type is defined, we can turn it into an enumeration simply as follows:

```
enum day;
```

There's also a convenience function `defenum` which defines the type and makes it enumerable in one go:

```
defenum day [sun,mon,tue,wed,thu,fri,sat];
```

In particular, this sets up the functions `day` and `ord` so that you can convert between members of the `day` type and the corresponding ordinals:

```
> ord sun;
0
> day (ans+3);
wed
```

You can also retrieve the type of an enumerated type member (or rather its enumeration function) with `enumof`:

```
> enumof sun;
day
> ans 5;
fri
```

Basic arithmetic, comparisons and arithmetic sequences also work as usual, provided that the involved members are all from the same enumeration:

```
> succ mon;
tue
> pred sat;
fri
> sun+3;
wed
> fri-2;
wed
> fri-tue;
3
> mon..fri;
[mon,tue,wed,thu,fri]
> sun:tue..sat;
[sun,tue,thu,sat]
```

```
> sat:fri..mon;
[sat,fri,thu,wed,tue,mon]
```

Note that given one member of the enumeration, you can use `enumof` to quickly enumerate *all* members of the type starting at the given member. Here's a little helper function which does this:

```
enumerate x::enum = iterwhile (typep ty) succ x when ty = enumof x end;
```

For instance:

```
> enumerate sun;
[sun,mon,tue,wed,thu,fri,sat]
```

Also note that `enum` silently skips elements which are already enumerated type members (no matter whether of the same or another type). Thus if you later add more elements to the `day` type, you can just call `enum` again to update the enumeration accordingly:

```
> succ sat;
sat+1
> type day doomsday;
> enum day;
()
> succ sat;
doomsday
```

2.4 Container Types

The standard library provides a variety of efficient container data structures for different purposes. These are all purely functional, i.e., immutable data structures implemented using different flavours of binary trees. This means that instead of modifying a data structure in-place, operations like insertion and deletion return a new instance of the container, keeping the previous instance intact. Nevertheless, all operations are performed efficiently, in logarithmic time where possible.

The container types are all implemented as abstract data structures, so client modules shouldn't rely on the internal representation. Each type provides a corresponding type tag (cf. *Type Tags* in the Pure Manual), as given in the description of each type, which can be used to match values of the type, e.g.:

```
shift a::array = rmfirst a;
```

All container types implement the equality predicates `==` and `~=` by recursively comparing their members. In addition, the dictionary, set and bag data structures also provide the other comparison predicates (`<`, `<=` etc.) which check whether one dictionary, set or bag is contained in another.

2.4.1 Arrays

The array.pure module implements an efficient functional array data structure which allows to access and update individual array members, as well as to add and remove elements at the beginning and end of an array. All these operations are carried out in logarithmic time.

type array
The array data type.

Imports

To use the operations of this module, add the following import declaration to your program:

```
using array;
```

Operations

emptyarray
return the empty array

array xs
create an array from a list `xs`

array2 xs
create a two-dimensional array from a list of lists

mkarray x n
create an array consisting of `n` x's

mkarray2 x (n,m)
create a two-dimensional array of `n*m` x's

arrayp x
check whether `x` is an array

a
size of `a`

a **!** i
return the `i`th member of `a`

a **!** (i,j)
two-dimensional subscript

null a
test whether `a` is the empty array

members a
list a
list of values stored in `a`

members2 a
list2 a
: list of members in a two-dimensional array

first a
last a
: first and last member of a

rmfirst a
rmlast a
: remove first and last member from a

insert a x
: insert x at the beginning of a

append a x
: append x to the end of a

update a i x
: replace the ith member of a by x

update2 a (i,j) x
: update two-dimensional array

Examples

Import the module:

```
> using array;
```

A one-dimensional array:

```
> let a::array = array (0.0:0.1..1.0);
> #a; members a;
11
[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
```

Indexing an array works in the usual way, using Pure's `!` operator. By virtue of the prelude, slicing an array with `!!` also works as expected:

```
> a!5;
0.5
> a!!(3..7);
[0.3,0.4,0.5,0.6,0.7]
```

Updating a member of an array produces a new array:

```
> let b::array = update a 1 2.0;
> members b;
[0.0,2.0,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
```

Two-dimensional arrays can be created with `array2` from a list of lists:

```
> let a2::array = array2 [[i,x | x = [u,v,w]] | i = 1..2];
> members2 a2;
[[(1,u),(1,v),(1,w)],[(2,u),(2,v),(2,w)]]
> a2!(1,2);
2,w
> a2!![(0,1),(1,2)];
[(1,v),(2,w)]
> a2!!(0..1,1..2);
[[(1,v),(1,w)],[(2,v),(2,w)]]
```

Here's how to convert an array to a Pure matrix:

```
> matrix $ members a;
{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0}
> matrix $ members2 a2;
{(1,u),(1,v),(1,w);(2,u),(2,v),(2,w)}
```

Converting back from a matrix to an array:

```
> let b2::array = array2 $ list2 {(1,u),(1,v),(1,w);(2,u),(2,v),(2,w)};
> members2 b2;
[[(1,u),(1,v),(1,w)],[(2,u),(2,v),(2,w)]]
```

2.4.2 Heaps

Heaps are a kind of priority queue data structure which allows quick (constant time) access to the smallest member, and to remove the smallest member and insert new elements in logarithmic time. Our implementation does not allow quick update of arbitrary heap members; if such functionality is required, bags can be used instead (see Sets and Bags).

Heap members *must* be ordered by the <= predicate. Multiple instances of the same element may be stored in a heap; however, the order in which equal elements are retrieved is not specified.

type heap
The heap data type.

Imports

To use the operations of this module, add the following import declaration to your program:

```
using heap;
```

Operations

emptyheap
return the empty heap

heap xs
: create a heap from a list `xs`

heapp x
: check whether `x` is a heap

h
: size of a heap

null h
: test whether `h` is the empty heap

members h
list h
: list the members of `h` in ascending order

first h
: the first (i.e., smallest) member of `h`

rmfirst h
: remove the first (i.e., smallest) member from `h`

insert h x
: insert `x` into `h`

Examples

```
> let h::heap = heap [5,1,3,11,3];
> members h;
[1,3,3,5,11]
> first h;
1
> members $ rmfirst h;
[3,3,5,11]
```

2.4.3 Dictionaries

The dict.pure module provides Pure's dictionary data types based on AVL trees. There are actually four different types to choose from, depending on whether you need ordered or hashed dictionaries and whether multiple values for the same key should be allowed or not.

type dict
: An ordered dictionary. This assumes an ordered key type, i.e., the predicate `<` must be defined on the keys.

type hdict
: A hashed dictionary which works with any (mixture of) key types but stores members in an apparently random order.

type mdict
An ordered dictionary, like dict, which allows multiple values to be associated with the same key.

type hmdict
A multi-valued dictionary, like mdict, but uses hashed keys like hdict.

type xdict
This is just an abstract supertype for matching any kind of dictionary provided by this module.

mdict and hmdict are also colloquially referred to as (ordered or hashed) *multidicts*. This implementation guarantees that different members for the same key are always kept in the order in which they were inserted, and this is also the order in which they will be retrieved by the members, keys, vals and indexing operations.

The usual comparison predicates (==, ~=, <=, < etc.) are defined on all dictionary types, where two dictionaries are considered "equal" (d1==d2) if they both contain the same key=>value pairs, and d1<=d2 means that d1 is a sub-dictionary of d2, i.e., all key=>value pairs of d1 are also contained in d2 (taking into account multiplicities in the multidict case). Ordered dictionaries compare keys using equality (assuming two keys a and b to be equal if neither a<b nor b<a holds), while hashed dictionaries check for syntactical equality (using ===). The associated values are compared using the == predicate if it is defined, falling back to syntactic equality otherwise.

The underlying AVL tree data structure can be found in the avltrees.pure module which is included in the library, but not to be invoked directly.

The AVL tree algorithm has its origin in the SWI-Prolog implementation of association lists. The original implementation was created by R. A. O'Keefe and updated for SWI-Prolog by Jan Wielemaker. For the original source see http://www.swi-prolog.org.

The port from SWI-Prolog and the deletion stuff (rmfirst, rmlast, delete) missing in the Prolog implementation was provided by Jiri Spitz. The generalization of the code to arbitrary combinations of ordered/hashed and single-/multi-valued keys was done by Albert Graef.

Imports

To use the operations of this module, add the following import declaration to your program:

```
using dict;
```

Operations

emptydict
emptyhdict
emptymdict
emptyhmdict
return an empty dictionary

dict xs
hdict xs
mdict xs
hmdict xs
: create a dictionary of the corresponding type either from a list `xs` of key-value pairs in the form `key=>value`, or from another dictionary; in the latter case the argument is converted to a dictionary of the desired target type

dictp d
hdictp d
mdictp d
hmdictp d
: check whether `x` is a dictionary of the corresponding type

mkdict y xs
mkhdict y xs
mkmdict y xs
mkhmdict y xs
: create a dictionary from a list of keys and a constant value

d1 **+** d2
: sum: `d1+d2` adds the members of `d2` to `d1`

d1 **-** d2
: difference: `d1-d2` removes the members of `d2` from `d1`

d1 ***** d2
: intersection: `d1*d2` removes the members *not* in `d2` from `d1`

d
: size of a dictionary (the number of members it contains)

d **!** x
: get the value from `d` by key `x`; in the case of a multidict this actually returns a list of values (which may be empty if `d` doesn't contain `x`)

null d
: test whether `d` is an empty dictionary

member d x
: test whether `d` contains a member with key `x`

members d
list d
: list the members of `d` (in ascending order for ordered dictionaries)

keys d
: list the keys of `d` (in ascending order for ordered dictionaries)

vals d
: list the values of `d`

first d

last d
: return the first and the last member of `d`, respectively

rmfirst d
rmlast d
: remove the first and the last member from `d`, respectively

insert d (x=>y)
update d x y
: insert `x=>y` into `d` (this always adds a new member in a multidict, otherwise it replaces an existing value if there is one); note that `update` is just a fully curried version of `insert`, so `update d x y` behaves exactly like `insert d (x=>y)`

delete d x
: remove `x` from `d` if present (in the multidict case, only the first member with the given key `x` is removed)

delete_val d (x=>y)
: remove a specific key-value pair `x=>y` from `d` if present (in the multidict case, only the first instance of `x=>y` is removed); please also see the notes below regarding this operation

delete_all d x
: remove all instances of `x` from `d` (in the non-multidict case, this is just the same as `delete`)

Note:

- The infix operators `+`, `-` and `*` work like the corresponding set and bag operations (see Sets and Bags), treating dictionaries as collections of `key=>val` pairs. You can mix arbitrary operand types with these operations, as well as with the comparison operations; the necessary conversions from less general dictionary types (ordered, single-valued) to more general types (hashed, multi-valued) are handled automatically.
- The `delete_val` function compares values using equality (`==`) if it is defined, falling back to syntactic equality (`===`) otherwise. If there is more than one instance of the given value under the given key, the first such instance will be removed (which, if `==` is defined on the values, may be any instance that compares equal, not necessarily an exact match).
- In the multidict case, `delete_val` may require linear time with respect to the number of different values stored under the given key. Since this operation is also needed to implement some other multidict operations like comparisons, difference and intersection, these may end up requiring quadratic running times in degenerate cases (i.e., if the majority of members happens to be associated with only very few keys).

Examples

A normal (ordered) dictionary:

```
> using dict;
> let d::dict = dict ["foo"=>77,"bar"=>99.1];
> keys d; vals d; members d;
["bar","foo"]
[99.1,77]
["bar"=>99.1,"foo"=>77]
```

Indexing a dictionary works in the usual way, using Pure's `!` operator. An `out_of_bounds` exception is thrown if the key is not in the dictionary:

```
> d!"foo";
77
> d!"baz";
<stdin>, line 5: unhandled exception 'out_of_bounds' while evaluating
'd!"baz"'
```

By virtue of the prelude, slicing a dictionary with `!!` also works as expected:

```
> d!!["foo","bar","baz"];
[77,99.1]
```

A hashed dictionary can be used with any key values, which are stored in a seemingly random order:

```
> let h::hdict = hdict [foo=>77,42=>99.1];
> keys h; vals h; members h;
[42,foo]
[99.1,77]
[42=>99.1,foo=>77]
> h!foo;
77
> h!!keys h;
[99.1,77]
```

Multidicts work in pretty much the same fashion, but allow more than one value for a given key to be stored in the dictionary. In this case, the indexing operation returns a list of all values for the given key, which may be empty if the key is not in the dictionary (rather than throwing an `out_of_bounds` exception):

```
> let d::mdict = mdict ["foo"=>77,"bar"=>99.1,"foo"=>99];
> d!"foo"; d!"baz";
[77,99]
[]
```

Slicing thus returns a list of lists of values here:

```
> d!!["foo","bar","baz"];
[[77,99],[99.1],[]]
```

To obtain a flat list you can just concatenate the results:

```
> cat $ d!!["foo","bar","baz"];
[77,99,99.1]
```

Hashed multidicts provide both key hashing and multiple values per key:

```
> let h::hmdict = hmdict [foo=>77,42=>99.1,42=>77];
> keys h; vals h; members h;
[42,42,foo]
[99.1,77,77]
[42=>99.1,42=>77,foo=>77]
> h!42;
[99.1,77]
```

There are also some set-like operations which allow you to add/remove the members (`key=>val` pairs) of one dictionary to/from another dictionary, and to compute the intersection of two dictionaries. For instance:

```
> let h1 = hmdict [a=>1,b=>2];
> let h2 = hmdict [b=>2,c=>3];
> members (h1+h2);
[a=>1,c=>3,b=>2,b=>2]
> members (h1-h2);
[a=>1]
> members (h1*h2);
[b=>2]
```

It's possible to mix dictionaries of different types in these operations. The necessary conversions are handled automatically:

```
> let h1 = hmdict [a=>1,b=>2];
> let h2 = hdict  [b=>3,c=>4];
> members (h1+h2);
[a=>1,c=>4,b=>2,b=>3]
```

Note that the result will always be promoted to the most general operand type in such cases (a hashed multidict in the above example). If this is not what you want, you'll have to apply the necessary conversions manually:

```
> members (hdict h1+h2);
[a=>1,c=>4,b=>3]
```

2.4.4 Sets and Bags

The set.pure module implements Pure's set data types based on AVL trees. These work pretty much like dictionaries (cf. Dictionaries) but only store keys (called "elements" or "members" here) without any associated data values. Hence sets provide membership tests like dictionaries, but no indexing operations.

There are four variations of this data structure to choose from, depending on whether the set members are ordered or hashed, and whether multiple instances of the same element are allowed (in this case the set is actually called a *multiset* or a *bag*).

type set

type bag
: These implement the ordered set types. They require that members be ordered, i.e., the predicate < must be defined on them.

type hset
type hbag
: These implement the hashed set types which don't require an order of the members. Distinct members are stored in an apparently random order.

type xset
: This is just an abstract supertype for matching any kind of set or bag provided by this module.

The usual comparison predicates (==, ~=, <=, < etc.) are defined on all set and bag types, where two sets or bags are considered "equal" (`m1==m2`) if they both contain the same elements, and `m1<=m2` means that `m1` is a subset or subbag of `m2`, i.e., all elements of `m1` are also contained in `m2` (taking into account multiplicities in the multiset case). Ordered sets and bags compare elements using equality (considering two elements `a` and `b` to be equal if neither `a<b` nor `b<a` holds), while hashed sets and bags check for syntactical equality (using ===).

The underlying AVL tree data structure can be found in the avltrees.pure module which is included in the library, but not to be invoked directly. The AVL tree algorithm has its origin in the SWI-Prolog implementation of association lists and was ported to Pure by Jiri Spitz, see Dictionaries for details.

Imports

To use the operations of this module, add the following import declaration to your program:

```
using set;
```

Operations

emptyset
emptybag
emptyhset
emptyhbag
: return an empty set or bag

set xs
bag xs
hset xs
hbag xs
: create a set or bag of the corresponding type from a list or another set or bag `xs`; in the latter case the argument is converted to a set or bag of the desired target type

setp x

bagp x
hsetp x
hbagp x
: check whether x is a set or bag of the corresponding type

m1 **+** m2
: union/sum: `m1+m2` adds the members of `m2` to `m1`

m1 **-** m2
: difference: `m1-m2` removes the members of `m2` from `m1`

m1 ***** m2
: intersection: `m1*m2` removes the members *not* in `m2` from `m1`

m
: size of a set or bag `m`

null m
: test whether `m` is an empty set or bag

member m x
: test whether `m` contains `x`

members m
list m
: list the members of `m` (in ascending order for ordered sets and bags)

first m
last m
: return the first and the last member of `m`, respectively

rmfirst m
rmlast m
: remove the first and the last member from `m`, respectively

insert m x
: insert `x` into `m` (replaces an existing element in the case of a set)

delete m x
: remove `x` from `m` (in the bag case, only the first instance of `x` is removed)

delete_all m x
: remove all instances of `x` from `m` (in the set case, this is just the same as `delete`)

Note: The infix operators (`+`, `-`, `*`, as well as the comparison operations) allow you to mix arbitrary operand types; the necessary conversions from less general set types (ordered, set) to more general types (hashed, multiset) are handled automatically.

Also note that in the case of sets, `+` is just the ordinary set union. There are basically two generalizations of this operation to bags, **multiset union** and **multiset sum**; `+` implements the *latter*. Thus, if a bag `m1` contains `k1` instances of an element `x` and a bag `m2` contains `k2` instances of `x`, then `m1+m2` contains `k1+k2` instances of `x` (rather than `max k1 k2` instances, which would be the case for multiset union). Multiset sum is probably more common in

practical applications, and also generalizes easily to multidicts (see Dictionaries). However, if multiset union is needed, it can easily be defined in terms of multiset sum as follows:

```
union m1 m2 = m1+(m2-m1);
```

Examples

Some basic set operations:

```
> let m::set = set [5,1,3,11,3];
> members m;
[1,3,5,11]
> map (member m) (1..5);
[1,0,1,0,1]
> members $ m+set (3..6);
[1,3,4,5,6,11]
> members $ m-set (3..6);
[1,11]
> members $ m*set (3..6);
[3,5]
```

The bag operations work in a similar fashion, but multiple instances are permitted in this case, and each instance counts as a separate member:

```
> let m::bag = bag [5,1,3,11,3];
> members m;
[1,3,3,5,11]
> members $ delete m 3;
[1,3,5,11]
> members $ insert m 1;
[1,1,3,3,5,11]
> members $ m+bag (3..6);
[1,3,3,3,4,5,5,6,11]
> members $ m-bag (3..6);
[1,3,11]
> members $ m*bag (3..6);
[3,5]
```

As already mentioned, operands of different types can be mixed with the infix operators; the necessary conversions are handled automatically. E.g., here's how you add a set to a bag:

```
> let m1::bag = bag [5,1,3,11,3];
> let m2::set = set (3..6);
> members (m1+m2);
[1,3,3,3,4,5,5,6,11]
```

Note that the result will always be promoted to the most general operand type in such cases (a bag in the above example). If this is not what you want, you'll have to apply the necessary conversions manually:

```
> members (set m1+m2);
[1,3,4,5,6,11]
```

If set members aren't ordered then you'll get an exception when trying to create an ordered set or bag from them:

```
> set [a,b,c];
<stdin>, line 5: unhandled exception 'failed_cond' while evaluating
'set [a,b,c]'
```

In such a case hashed sets and bags must be used instead. These work analogously to the ordered sets and bags, but distinct members are stored in an apparently random order:

```
> members $ hset [a,b,c] * hset [c,d,e];
[c]
> members $ hbag [a,b,c] + hbag [c,d,e];
[a,c,c,b,d,e]
```

2.5 System Interface

This module offers some useful system routines, straight from the C library, as well as some convenience functions for wrapping these up in Pure. Even the "purest" program needs to do some basic I/O every once in a while, and this module provides the necessary stuff to do just that. The operations provided in this module should work (if necessary by a suitable emulation) on all supported systems. Most of the following functions are extensively documented in the C library manual pages, so we concentrate on the Pure-specific aspects here.

2.5.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using system;
```

Some functions of the system interface are provided in separate modules; see Regex Matching, Additional POSIX Functions and Option Parsing.

2.5.2 Errno and Friends

errno
set_errno n
perror msg
strerror n

This value and the related routines are indispensable to give proper diagnostics when system calls fail for some reason. Note that, by its very nature, `errno` is a fairly volatile value, don't expect it to survive a return to the command line in interactive sessions.

Example:

```
> using system;
> fopen "junk" "r", perror "junk";
junk: No such file or directory
fopen "junk" "r"
```

2.5.3 POSIX Locale

setlocale category locale
: Set or retrieve the current locale.

Details are platform-specific, but you can expect that at least the categories LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC and LC_TIME are defined, as well as the following values for the locale parameter: "C" or "POSIX" (the default POSIX locale), "" (the system default locale), and NULL, to just query the current locale.

Other string values which can be passed as the locale argument depend on the implementation, please check your local setlocale(3) documentation for details. If locale is not NULL, the current locale is changed accordingly. The return value is the new locale, or the current locale when passing NULL for the locale parameter. In either case, the string returned by setlocale is such that it can be passed to setlocale to restore the same locale again. In case of an error, setlocale fails (rather than returning a null pointer).

Please note that calling this function alters the Pure interpreter's idea of what the current locale is. When the interpreter starts up, it always sets the default system locale. Unless your scripts rely on a specific encoding, setting the locale to either "C" or "" should always be safe.

Example:

```
> setlocale LC_ALL NULL;
"en_US.UTF-8"
```

2.5.4 Signal Handling

trap action sig
: Establish or remove Pure signal handlers.

The action parameter of trap can be one of the predefined integer values SIG_TRAP, SIG_IGN and SIG_DFL. SIG_TRAP causes the given signal to be handled by mapping it to a Pure exception of the form signal sig. SIG_IGN ignores the signal, SIG_DFL reverts to the system's default handling. See show -g SIG* for a list of known signal values on your system.

Note: When the interpreter runs interactively, most standard termination signals (SIGINT, SIGTERM, etc.) are already set up to report corresponding Pure exceptions; if this is not desired, you can use trap to either ignore these or revert to the default handlers instead.

See *Exception Handling* in the Pure Manual for details and examples.

2.5.5 Time Functions

The usual date/time functions from the C library are all provided. This includes some functions to retrieve wallclock and cpu time which usually offer much better resolution than the venerable `time` function.

`time`
: Reports the current time in seconds since the **epoch**, 00:00:00 UTC, Jan 1 1970. The result is always a bigint (in fact, the `time` value is already 64 bit on many OSes nowadays).

`gettimeofday`
: Returns wallclock time as seconds since the epoch, like `time`, but theoretically offers resolutions in the microsec range (actual resolutions vary, but are usually in the msec range for contemporary systems). The result is returned as a double value (which also limits precision). This function may actually be implemented through different system calls, depending on what's available on the host OS.

`clock`
: Returns the current CPU (not wallclock) time since an arbitrary point in the past, as a machine int. The number of "ticks" per second is given by the `CLOCKS_PER_SEC` constant. Note that this value will wrap around approximately every 72 minutes.

`sleep` t
`nanosleep` t
: Suspend execution for a given time interval in seconds. `sleep` takes integer (int/bigint) arguments only and uses the `sleep()` system function. `nanosleep` also accepts double arguments and theoretically supports resolutions down to 1 nanosecond (again, actual resolutions vary). This function may actually be implemented through different system calls, depending on what's available on the host OS. Both functions usually return zero, unless the sleep was interrupted by a signal, in which case the time remaining to be slept is returned.

Examples:

```
> time,sleep 1,time;
1270241703L,0,1270241704L
> gettimeofday,nanosleep 0.1,gettimeofday;
1270241709.06338,0.0,1270241709.16341
```

Here's a little macro which lets you time evaluations:

```
def timex x = y,(t2-t1)/CLOCKS_PER_SEC when
  t1 = clock; y = x; t2 = clock;
end;
```

Example:

```
> timex (foldl (+) 0 (1..100000));
705082704,0.07
```

tzset
Initialize timezone information.

variable tzname
variable timezone
variable daylight
The timezone information.

The `tzset` function calls the corresponding routine from the C library and initializes the (Pure) variables `tzname`, `timezone` and `daylight` accordingly. See the tzset(3) manual page for details. This routine is also called automatically when the system module is loaded, so you only have to invoke it to get up-to-date information after changes to the locale or the timezone. Example:

```
> tzset;
()
> tzname, timezone, daylight;
["CET","CEST"],-3600,1
> tzname!daylight;
"CEST"
```

The following functions deal with date/time values in string and "broken-down" time format. See the ctime(3), gmtime(3), localtime(3), mktime(3), asctime(3), strftime(3) and strptime(3) manual pages for details.

ctime t
Convert a time value as returned by the `time` function to a string in local time.

gmtime t
localtime t
Convert a time value to UTC or local time in "broken-down" form (a static pointer to a `tm` struct containing a bunch of `int` fields) which can then be passed to the `asctime` and `strftime` functions, or to `int_matrix` if you want to convert the data to a matrix; see the example below.

mktime tm
Converts broken-down time to a time value (seconds since the epoch). As with `time`, the result is always a bigint.

asctime tm
strftime format tm
Format broken-down time as a string. `strftime` also uses a format string supplied by the user, see below for a list of the most important conversion specifiers.

strptime s format tm
Parse a date/time string `s` according to the given format (using more or less the same format specifiers as the `strftime` function) and store the broken-down time result in the given `tm` struct. This function may fail, e.g., if `strptime` finds an error in the format string. Otherwise it returns the part of the string which wasn't processed, see the example below.

Examples:

```
> let t = time; t;
1270239790L
> let tm = localtime t; tm;
#<pointer 0x7ff97ecbdde0>
> mktime tm;
1270239790L
> asctime tm;
"Fri Apr  2 22:23:10 2010\n"
> int_matrix 9 tm;
{10,23,22,2,3,110,5,91,1}
> strftime "%c" tm;
"Fri 02 Apr 2010 10:23:10 PM CEST"
> strptime ans "%c" tm, int_matrix 9 tm;
"CEST",{10,23,22,2,3,110,5,91,1}
```

In the above example, `strptime` was given a static pointer to a `tm` struct returned by `localtime`. This always works, but in some situations it may be preferable to allocate dynamic storage instead. This storage should be properly initialized (zeroed out) before passing it to `strptime`, since `strptime` only stores the values specified (at least in principle; please consult your local C library documentation for details). Also note that while POSIX only specifies nine `int` fields in a `tm` struct, depending on the host operating system the struct may contain additional public and private fields. The actual size of a `tm` struct is given by the `SIZEOF_TM` constant, so a safe way to allocate suitable dynamic storage for the `strptime` function is as follows:

```
> let tm = pointer_cast "int*" $ calloc 1 SIZEOF_TM;
> strptime "4/2/10" "%D" tm, int_matrix 9 tm;
"",{0,0,0,2,3,110,5,91,0}
```

Instead of explicitly allocating dynamic storage and converting it to a Pure matrix later, you can also invoke `strptime` directly with an int matrix of sufficient size:

```
> let tm = imatrix (SIZEOF_TM div SIZEOF_INT + 1);
> strptime "4/2/10" "%D" tm, take 9 tm;
"",{0,0,0,2,3,110,5,91,0}
```

Last but not least, to make calling `strptime` more convenient, you can supply your own little wrapper function which takes care of allocating the storage, e.g.:

```
mystrptime s format = s,take 9 tm when
  tm = imatrix (SIZEOF_TM div SIZEOF_INT + 1);
  s = strptime s format tm;
end;

> mystrptime "4/2/10" "%D";
"",{0,0,0,2,3,110,5,91,0}
```

Here is a list of some common format specifiers which can be used with the `strftime` and `strptime` routines. These are all specified by POSIX and should thus be available on most platforms. Note that many more formats are usually supported than what is listed here, so please consult your local manual pages for the complete list.

- `%d`, `%m`, `%y`: Day of the month, month and year as decimal two-digit numbers.
- `%Y`: The year as a four-digit number which includes the century.
- `%H`, `%M`, `%S`: Hours (range `00` to `23`), minutes and seconds as decimal two-digit numbers.
- `%I`: The hours on a 12-hour clock (range `01` to `12`).

The following formats are locale-dependent:

- `%a`, `%A`: Abbreviated and full weekday name.
- `%b`, `%B`: Abbreviated and full month name.
- `%p`: AM or PM. `%P` is the same in lowercase (`strftime` only).

There are also some useful meta-formats which specify various combinations of the above:

- `%c`: The preferred date and time representation for the current locale.
- `%D`: The American date format (`%m/%d/%y`).
- `%F`: The ISO 8601 date format (`%Y-%m-%d`). (This is generally supported by `strftime` only, but `strptime` from GNU libc has it.)
- `%r`: The time in AM/PM notation (`%I:%M:%S %p`).
- `%R`: The time in 24-hour notation (`%H:%M`).
- `%T`: The time in 24-hour notation, including seconds (`%H:%M:%S`).

In addition, `%%` denotes a literal `%` character, `%n` newlines and `%t` tabs. (For `strptime` the latter two are synonymous and match arbitrary whitespace.)

Windows users should note that `strptime` isn't natively supported there. A basic emulation is provided by the Pure runtime, but at present this only supports the C locale.

2.5.6 Process Functions

The following process functions are available on all systems. (Some additional process-related functions such as `fork`, `kill`, `wait` and `waitpid` are available in the `posix` module, see Additional POSIX Functions.)

`system` cmd
: Execute a shell command.

`execv` prog argv
`execvp` prog argv
`execve` prog argv envp
: Execute a new process. `prog` denotes the name of the executable to be run, `argv` the argument vector (which repeats the program name in the first component), and `envp` a vector of environment strings of the form `"var=value"`. The `execv` function executes the program `prog` exactly as given, while `execvp` also performs a path search. The `execve` function is like `execv`, but also specifies an environment to be passed to the process. In either case, the new process replaces the current process. For convenience,

both `argv` and `envp` can be specified as a Pure string vector or a list, which is automatically translated to the raw, `NULL`-terminated C string vectors (i.e., `char**`) required by the underlying C functions.

spawnv mode prog argv
spawnvp mode prog argv
spawnve mode prog argv envp
Spawn a new child process. These work like the corresponding MS Windows functions; on Un*x systems this functionality is implemented using a combination of `fork` and `execv`. The arguments are the same as for the `execv` functions, except that there's an additional `mode` argument which specifies how the process is to be executed: `P_WAIT` waits for the process to finish, after which `spawnv` returns with the exit status of the terminated child process; `P_NOWAIT` makes `spawnv` return immediately, returning the process id; and `P_OVERLAY` causes the child process to replace its parent, just like with `execv`. (On Windows, there's an additional `P_DETACH` flag which works like `P_NOWAIT` but also turns the child process into a background task.)

Note that, in addition, the prelude provides the `exit` function which terminates the program with a given exit code, cf. Other Special Primitives.

Examples:

```
> system "pwd";
/home/ag/svn/pure-lang/trunk/pure/lib
0
> spawnvp P_WAIT "pwd" ["pwd"];
/home/ag/svn/pure-lang/trunk/pure/lib
0
> spawnv P_WAIT "/bin/sh" ["/bin/sh","-c","pwd"];
/home/ag/svn/pure-lang/trunk/pure/lib
0
```

2.5.7 Basic I/O Interface

Note that this module also defines the standard I/O streams `stdin`, `stdout` and `stderr` as variables on startup. These are ready to be used with the operations described below. Also note that for convenience some of the following routines are actually Pure wrappers, rather than just providing the raw C library routines.

variable stdin
variable stdout
variable stderr
The standard I/O streams.

fopen name mode
popen cmd mode
Open a file or a pipe. These take care of closing a file object automagically when it's garbage-collected, and fail (instead of returning a null pointer) in case of error, so that you can provide any desired error handling simply by adding suitable equations.

fdopen fd mode
: Associates a file object with a given existing file descriptor. Otherwise works like `fopen`, so the resulting file is closed automatically when it's garbage-collected.

freopen path mode fp
: Reopens a file object. The existing file object is closed. Otherwise works like `fopen`, so the resulting file is closed automatically when it's garbage-collected.

fclose fp
pclose fp
: Close a file or a pipe.

tmpfile
: Creates a unique temporary file (opened in `"w+b"` mode) which gets deleted automatically when it is closed or the file object gets garbage-collected.

feof fp
ferror fp
clearerr fp
: Check the end-of-file and error bits. `clearerr` clears the error bit.

fileno fp
: Returns the file descriptor associated with the given file.

fflush fp
: Flushes the given file (or all open files if `fp` is `NULL`).

fgets fp
gets
: Pure wrappers for the C `fgets` and `gets` functions which handle the necessary buffering automatically.

fget fp
: A variation of `fgets` which slurps in an entire text file at once.

fputs s fp
puts s
: Output a string to the given file or `stdout`, respectively. These are just the plain C functions. Note that `puts` automatically adds a newline, while `fputs` doesn't. Hmm.

fread ptr size nmemb fp
fwrite ptr size nmemb fp
: Binary read/writes. Here you'll have to manage the buffers yourself. See the corresponding manual pages for details.

fseek fp offset whence
ftell fp
rewind fp
: Reposition the file pointer and retrieve its current value. The constants `SEEK_SET`, `SEEK_CUR` and `SEEK_END` can be used for the `whence` argument of `fseek`. The call `rewind fp` is equivalent to `fseek fp 0 SEEK_SET` (except that the latter also returns a result code). See the corresponding manual pages for details.

setbuf fp buf
setvbuf fp buf mode size

Set the buffering of a file object, given as the first argument. The second argument specifies the buffer, which must be a pointer to suitably allocated memory or NULL. The mode argument of setvbuf specifies the buffering mode, which may be one of the predefined constants _IONBF, _IOLBF and _IOFBF denoting no buffering, line buffering and full (a.k.a. block) buffering, respectively; the size argument denotes the buffer size.

For setbuf, the given buffer must be able to hold BUFSIZ characters, where BUFSIZ is a constant defined by this module. `setbuf fp buf` is actually equivalent to the following call (except that setvbuf also returns an integer return value):

```
setvbuf fp buf (if null buf then _IONBF else _IOFBF) BUFSIZ
```

Please see the setbuf(3) manual page for details.

Examples:

```
> puts "Hello, world!";
Hello, world!
14

> map fileno [stdin,stdout,stderr];
[0,1,2]

> let fp = fopen "/etc/passwd" "r";
> fgets fp;
"at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash\n"
> fgets fp;
"avahi:x:103:104:User for Avahi:/var/run/avahi-daemon:/bin/false\n"
> ftell fp;
121L
> rewind fp;
()
> fgets fp;
"at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash\n"

> split "\n" $ fget $ popen "ls *.pure" "r";
["array.pure","dict.pure","getopt.pure","heap.pure","math.pure",
"matrices.pure","prelude.pure","primitives.pure","quasiquote.pure",
"set.pure","strings.pure","system.pure",""]
```

C-style formatted I/O is provided through the following wrappers for the C printf and scanf functions. These wrapper functions take or return a tuple of values and are fully type-safe, so they should never segfault. All basic formats derived from %cdioux, %efg, %s and %p are supported, albeit without the standard length modifiers such as h and l, which aren't of much use in Pure. (However, in addition to C printf and scanf, the Pure versions also support the modifiers Z and R of the GMP and MPFR libraries, which are used for converting multiprecision integer and floating point values, as shown in the examples below.)

printf format args

fprintf fp format args
: Print a formatted string to `stdout` or the given file, respectively. Normally, these functions return the result of the underlying C routines (number of characters written, or negative on error). However, in case of an abnormal condition in the wrapper function, such as argument mismatch, they will throw an exception. (In particular, an `out_of_bounds` exception will be thrown if there are not enough arguments for the given format string.)

sprintf format args
: Print a formatted string to a buffer and return the result as a string. Note that, unlike the C routine, the Pure version just returns the string result in the case of success; otherwise, the error handling is the same as with `printf` and `fprintf`. The implementation actually uses the C routine `snprintf` for safety, and a suitable output buffer is provided automatically.

scanf format
fscanf fp format
: Read formatted input from `stdin` or the given file, respectively. These normally return a tuple (or singleton) with the converted values. An exception of the form `scanf_error ret`, where `ret` is the tuple of successfully converted values (which may be less than the number of requested input items), is thrown if end-of-file was met or another error occurred while still reading. The handling of other abnormal conditions is analogous to `printf` et al. Also note that this implementation doesn't accept any of the standard length modifiers; in particular, floating point values will *always* be read in double precision and you just specify `e`, `g` etc. for these. The "assignment suppression" flag `*` is understood, however; the corresponding items will not be returned.

sscanf s format
: This works exactly like `fscanf`, but input comes from a string (first argument) rather than a file.

Examples:

```
> do (printf "%s%d\n") [("foo",5),("catch",22)];
foo5
catch22
()
> sscanf "foo 5 22" "%s %d %g";
"foo",5,22.0
```

As mentioned above, special argument formats are provided for bigints and multiprecision floats:

```
> sscanf "a(5) = 1234" "a(%d) = %Zd";
5,1234L
> sprintf "a(%d) = %Zd" ans;
"a(5) = 1234"

> using mpfr;
> mpfr_set_default_prec 113;
()
```

```
> printf "pi = %0.30Rg\n" (4*atan (mpfr 1));
pi = 3.14159265358979323846264338328
37
```

There are a number of other options for these conversions, please check the GMP and MPFR documentation for details.

Note: In contrast to bigints, multiprecision floats aren't directly supported by the Pure language. If you would like to use these numbers, you'll have to install the `mpfr` addon module which is not included in the standard library yet. Also note that, at the time of this writing, MPFR only provides formatted output, so multiprecision floats are not supported by the `scanf` functions. To work around this limitation, it is possible to read the number as a string and then convert it using the `mpfr` function.

2.5.8 Stat and Friends

`stat` path
: Return information about the given file. This is a simple wrapper around the corresponding system call, see the stat(2) manual page for details. The function returns a tuple with the most important fields from the `stat` structure, in this order: `st_dev`, `st_ino`, `st_mode`, `st_nlink`, `st_uid`, `st_gid`, `st_rdev`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. Among these, `st_mode`, `st_nlink`, `st_uid` and `st_gid` are simple machine integers, the rest is encoded as bigints (even on 32 bit platforms).

`lstat` path
: Return information about the given symbolic link (rather than the file it points to). On systems where this function isn't supported (e.g., Windows), `lstat` is identical to `stat`.

`fstat` fp
: Return information about the given file object. Same as `stat`, but here the file is given as a file pointer created with `fopen` (see Basic I/O Interface above). Note that the corresponding system function actually takes a file descriptor, so the Pure implementation is equivalent to the C call `fstat(fileno(fp))`. This function might not be supported on all platforms.

For average applications, the most interesting fields are `st_mode` and `st_size`, which can be retrieved with `stat filename!![2,7]`. Note that to facilitate access to the `st_mode` field, the usual masks and bits for file types (`S_IFMT`, `S_IFREG`, etc.) and permissions (`S_ISUID`, `S_ISGID`, `S_IRWXU`, etc.) are defined as constants by this module. Use the command `show -g S_*` in the interpreter to get a full list of these. Other interesting fields are `st_atime`, `st_mtime` and `st_ctime`, which can be accessed using `stat filename!!(8..10)`. The values of these fields are the times of last access, last modification and creation, respectively, which can be decoded using the appropriate time functions like `ctime` or `strftime`, see Time Functions.

Examples:

```
> stat "/etc/passwd";
64773L,9726294L,33188,1,0,0,0L,1623L,1250373163L,1242692339L,1242692339L
> stat "/etc/passwd"!7;                              // file size
1623L
> strftime "%c" $ localtime $ stat "/etc/passwd"!10;   // creation time
"Tue 19 May 2009 02:18:59 AM CEST"
> sprintf "0%o" $ stat "/etc/passwd"!2 and not S_IFMT; // permissions
"0644"
> stat "/etc/passwd"!2 and S_IFMT == S_IFREG; // this is a regular file
1
> stat "/etc"!2 and S_IFMT == S_IFDIR;        // this is a directory
1
```

2.5.9 Reading Directories

readdir name
: Read the contents of the given directory and return the names of all its entries as a list.

Example:

```
> readdir "/home";
["ag",".",".."]
```

2.5.10 Shell Globbing

fnmatch pat s flags
: Returns a simple truth value (1 if `pat` matches `s`, 0 if it doesn't), instead of an error code like the C function.

glob pat flags
: Returns a Pure list with the matches (unless there is an error in which case the integer result code of the underlying C routine is returned).

The available flag values and glob error codes are available as symbolic `FNM_*` and `GLOB_*` constants defined as variables in the global environment. See the fnmatch(3) and glob(3) manpages for the meaning of these.

Example:

```
> glob "*.pure" 0;
["array.pure","dict.pure","getopt.pure","heap.pure","math.pure",
"matrices.pure","prelude.pure","primitives.pure","set.pure",
"strings.pure","system.pure"]
```

2.5.11 Regex Matching

Please note that, as of Pure 0.48, this part of the system interface is not included in the system module any more, but is provided as a separate regex module which can be used indepen-

dently of the system module. To use the operations of this module, add the following import declaration to your program:

```
using regex;
```

Since the POSIX regex functions (`regcomp` and `regexec`) have a somewhat difficult calling sequence, this module provides a couple of rather elaborate high-level wrapper functions for use in Pure programs. These are implemented in terms of a low-level interface provided in the runtime. (The low-level interface isn't documented here, but these functions are also callable if you want to create your own regular expression engines in Pure. You might wish to take a look at the implementation of the high-level functions in regex.pure to see how this can be done.)

regex pat cflags s eflags

Compiles and matches a regex in one go, and returns the list of submatches (if any).

Parameters

- **pat** (*string*) – the regular expression pattern
- **cflags** (*int*) – the compilation flags (bitwise or of any of the flags accepted by regcomp(3))
- **s** (*string*) – the subject string to be matched
- **eflags** (*int*) – the matching execution flags (bitwise or of any of the flags accepted by regexec(3))

Symbolic `REG_*` constants are provided for the different flag values, see the regcomp(3) manpage for an explanation of these. Two particularly important compilation flags (to be included in the `cflags` argument) are `REG_NOSUB`, which prevents submatches to be computed, and `REG_EXTENDED`, which switches `regex` from "basic" to "extended" regular expressions so that it understands all the regular expression elements of egrep(1) in the pattern argument.

Depending on the flags and the outcome of the operation, the result of this function can take one of the following forms:

- `regerr code msg`: This indicates an error during compilation of the pattern (e.g., if there was a syntax error in the pattern). `code` is the nonzero integer code returned by `regcomp`, and `msg` is the corresponding error message string, as returned by `regerror`. You can redefine the `regerr` function as appropriate for your application (e.g., if you'd like to print an error message or throw an exception).
- `0` or `1`: Just a truth value indicates whether the pattern matched or not. This will be the form of the result if the `REG_NOSUB` flag was specified for compilation, indicating that no submatch information is to be computed.
- `0` (indicating no match), or `1` (indicating a successful match), where the latter value is followed by a tuple of `(pos,substr)` pairs for each submatch. This will be the form of the result only if the `REG_NOSUB` flag was *not* specified for compilation, so that submatch information is available.

Note that, according to POSIX semantics, a return value of 1 does *not* generally mean that the entire subject string was matched, unless you explicitly tie the pattern to the beginning (^) and end ($) of the string.

If the result takes the latter form, each `(pos,substr)` pair indicates a portion of the subject string which was matched; `pos` is the position at which the match starts, and `substr` is the substring (starting at position `pos`) which was matched. The first `(pos,substr)` pair always indicates which portion of the string was matched by the entire pattern, the remaining pairs represent submatches for the parenthesized subpatterns of the pattern, as described on the regcomp(3) manual page. Note that some submatches may be empty (if they matched the empty string), in which case a pair `(pos,"")` indicates the (nonnegative) position `pos` where the subpattern matched the empty string. Other submatches may not participate in the match at all, in which case the pair `(-1,"")` is returned.

The following helper functions are provided to analyze the result returned by `regex`.

`reg_result` res
: Returns the result of a `regex` call, i.e., a `regerr` term if compilation failed, and a flag indicating whether the match was successful otherwise.

`reg_info` res
: Returns the submatch info if any, otherwise it returns `()`.

`reg` n info
: Returns the `n`th submatch of the given submatch info, where `info` is the result of a `reg_info` call.

`regs` info
: Returns all valid submatches, i.e., the list of all triples `(n,p,s)` for which `reg n == (p,s)` with `p>=0`.

In addition, the following convenience functions are provided to perform global regex searches, to perform substitutions, and to tokenize a string according to a given delimiter regex.

`regexg` f pat cflags s eflags
: Perform a global regular expression search. This routine will scan the entire string for (non-overlapping) instances of the pattern, applies the given function `f` to the `reg_info` for each match, and collects all results in a list. Note: Never specify the `REG_NOSUB` flag with this function, it needs the submatch info.

`regexgg` f pat cflags s eflags
: This works like `regexg`, but allows overlapping matches.

`regsub` f pat cflags s eflags
: Replaces all non-overlapping instances of a pattern with a computed substitution string. To these ends, the given function `f` is applied to the `reg_info` for each match. The result string is then obtained by concatenating `f info` for all matches, with the unmatched portions of the string in between. To make this work, `f` must always return a string value; otherwise, `regsub` throws a `bad_string_value` exception.

regsplit pat cflags s eflags
: Splits a string into constituents delimited by substrings matching the given pattern.

Please note that these operations all operate in an eager fashion, i.e., they process the entire input string in one go. This may be unwieldy or at least inefficient for huge amounts of text. As a remedy, the following lazy alternatives are available:

regexgs f pat cflags s eflags
regexggs f pat cflags s eflags
regsplits pat cflags s eflags
: These work like `regexg`, `regexgg` and `regsplit` above, but return a stream result which enables you to process the matches one by one, using "call by need" evaluation.

Basic Examples

Let's have a look at some simple examples:

```
> let pat = "[[:alpha:]][[:alnum:]]*";
> let s = "1var foo 99 BAR $%&";
```

Simple match:

```
> regex pat 0 s 0;
1,1,"var"
```

Same without match info:

```
> regex pat REG_NOSUB s 0;
1
```

Global match, return the list of all matches:

```
> regexg id pat 0 s 0;
[(1,"var"),(5,"foo"),(12,"BAR")]
```

Same with overlapping matches:

```
> regexgg id pat 0 s 0;
[(1,"var"),(2,"ar"),(3,"r"),(5,"foo"),(6,"oo"),(7,"o"),(12,"BAR"),
(13,"AR"),(14,"R")]
```

Note that `id` (the identity function) in the examples above can be replaced with an arbitrary function which processes the matches. For instance, if we only want the matched strings instead of the full match info:

```
> regexg (!1) pat 0 s 0;
["var","foo","BAR"]
```

Lazy versions of both `regexg` and `regexgg` are provided which return the result as a stream instead. These can be processed in a "call by need" fashion:

```
> regexgs id pat 0 s 0;
(1,"var"):#<thunk 0x7fb1b7976750>
> last ans;
12,"BAR"
```

Let's verify that the processing is really done lazily:

```
> using system;
> test x = printf "got: %s\n" (str x) $$ x;
> let xs = regexgs test pat 0 s 0;
got: 1,"var"
> xs!1;
got: 5,"foo"
5,"foo"
> last xs;
got: 12,"BAR"
12,"BAR"
```

As you can see, the first match is produced immediately, while the remaining matches are processed as the result stream is traversed. This is most useful if you have to deal with bigger amounts of text. By processing the result stream in a piecemeal fashion, you can avoid keeping the entire result list in memory. For instance, compare the following:

```
> let s2 = fget $ fopen "system.pure" "r";
> stats -m
> #regexg id pat 0 s2 0;
7977
0.18s, 55847 cells
> #regexgs id pat 0 s2 0;
7977
0.12s, 20 cells
```

Regex Substitutions and Splitting

We can also perform substitutions on matches:

```
> regsub (sprintf "<%d:%s>") pat 0 s 0;
"1<1:var> <5:foo> 99 <12:BAR> $%&"
```

Or split a string using a delimiter pattern (this uses an egrep pattern):

```
> let delim = "[[:space:]]+";
> regsplit delim REG_EXTENDED s 0;
["1var","foo","99","BAR","$%&"]
> regsplit delim REG_EXTENDED "The   quick brown    fox" 0;
["The","quick","brown","fox"]
```

The `regsplit` operation also has a lazy variation:

```
> regsplits "[[:space:]]+" REG_EXTENDED "The   quick brown    fox" 0;
"The":#<thunk 0x7fb1b79775b0>
```

```
> last ans;
"fox"
```

Empty Matches

Empty matches are permitted, too, subject to the constraint that at most one match is reported for each position (which also prevents looping). And of course an empty match will only be reported if nothing else matches. For instance:

```
> regexg id "" REG_EXTENDED "foo" 0;
[(0,""),(1,""),(2,""),(3,"")]
> regexg id "o*" REG_EXTENDED "foo" 0;
[(0,""),(1,"oo"),(3,"")]
> regexgg id "o*" REG_EXTENDED "foo" 0;
[(0,""),(1,"oo"),(2,"o"),(3,"")]
```

This also works when substituting or splitting:

```
> regsub (cst " ") "" REG_EXTENDED "some text" 0;
" s o m e   t e x t "
> regsub (cst " ") " ?" REG_EXTENDED "some text" 0;
" s o m e  t e x t "
> regsplit "" REG_EXTENDED "some text" 0;
["","s","o","m","e"," ","t","e","x","t",""]
> regsplit " ?" REG_EXTENDED "some text" 0;
["","s","o","m","e","","t","e","x","t",""]
```

Submatches

Parenthesized subexpressions in a pattern yield corresponding submatch information, which is useful if we need to retrieve the text matched by a given subexpression. For instance, suppose we want to parse environment lines, such as those returned by the shell's `set` command. These can be dissected using the following regex:

```
> const env_pat = "^([^=]+)=(.*)$";
> const env_flags = REG_EXTENDED or REG_NEWLINE;
> regex env_pat env_flags "SHELL=/bin/sh" 0;
1,0,"SHELL=/bin/sh",0,"SHELL",6,"/bin/sh"
```

Note that we again used an extended regex here, and we also added the `REG_NEWLINE` flag so that we properly deal with multiline input. The desired information is in the 4th and 6th element of the submatch info, we can retrieve that as follows:

```
> parse_env s = regexg (\info -> info!3 => info!5) env_pat env_flags s 0;
> parse_env "SHELL=/bin/sh\nHOME=/home/bar\n";
["SHELL"=>"/bin/sh","HOME"=>"/home/bar"]
```

We can get hold of the real process environment as follows:

```
> using system;
> let env = parse_env $ fget $ popen "set" "r";
> #env;
109
> head env;
"BASH"=>"/usr/bin/sh"
```

Just for the fun of it, let's convert this to a record, providing easy random access to the environment variables:

```
> let env = record env;
> env!!["SHELL","HOME"];
{"/bin/bash","/home/ag"}
```

2.5.12 Additional POSIX Functions

Platforms: Mac, Unix The posix module provides some additional POSIX functions not available on all supported systems. (In particular, none of these functions are provided on MS Windows.) You can load this module in addition to the system module if you need the additional functionality. To use the operations of this module, add the following import declaration to your program:

```
using posix;
```

The following operations are provided. Please see the appropriate POSIX manual pages for a closer description of these functions.

fork
: Fork a new process.

getpid
getppid
: Get the process id of the current process and its parent process, respectively.

wait status
waitpid pid status options
: Wait for any child process, or the given one. The `status` argument must be a pointer to an `int` value, which is used to return the status of the child process.

kill pid sig
: Send the given signal to the given process.

raise sig
: Raise the given signal in the current process.

pause
: Sleep until a signal is caught.

2.5.13 Option Parsing

This is a quick-and-dirty replacement for the GNU getopt functions, ported from the Q library. To use the operations of this module, add the following import declaration to your program:

```
using getopt;
```

The following operation is provided:

getopt opts args
: Parse options as given by `opts` in the command line arguments `args`, return the parsed options along with a list of the remaining (non-option) command line arguments.

The `getopt` function takes two arguments: `opts`, a list of option descriptions in the format described below, and `args`, a list of strings containing the command line parameters to be parsed for options. The result is a pair `(opts_return,args_return)` where `opts_return` is a list of options and their values, and `args_return` is the list of remaining (non-option) arguments. Options are parsed using the rules of GNU getopt(1). If an invalid option is encountered (unrecognized option, missing or extra argument, etc.), `getopt` throws the offending option string as an exception.

The `opts_return` value is a list of "hash pairs" `opt=>val` where `opt` is the (long) option name (as given by the `long_opt` field given in the `opts` argument, see below) and `val` is the corresponding value (`()` if none). Note that this format is ready to be passed to the `dict` or `hdict` function, cf. Dictionaries, which makes it easy to retrieve option values or check for the presence of options. (As of Pure 0.41, you can also just convert the list to a record and employ the record functions to access the option data, cf. Record Functions.)

The `opts` argument of `getopt` must be a list of triples `(long_opt, short_opt, flag)`, where `long_opt` denotes the long option, `short_opt` the equivalent short option, and `flag` is one of the symbolic integer values `NOARG`, `OPTARG` and `REQARG` which specifies whether the option has no argument, an optional argument or a required argument, respectively. Either `long_opt` or `short_opt` should be a string value of the form `"--abc"` or `"-x"`, respectively. Note that since the `long_opt` value is always used to denote the corresponding option in the `opts_return` list, you always have to specify a sensible value for that field. If no separate long option name is needed, you can specify the same value as in the `short_opt` field, or some other convenient value (e.g., an integer) which designates the option. Conversely, to indicate that an option has no short option equivalent, simply specify an empty option string for the `short_opt` field.

Examples:

```
> let opts = [("--help", "-h", NOARG),       // no argument
>             ("--version", "", NOARG),      // no short option
>             ("--filename", "-f", REQARG),  // required argument
>             ("--count", "-n", OPTARG)];    // optional argument
> getopt opts ["foo", "-h", "--filename", "bar", "-n0", "baz"];
["--help"=>(),"--filename"=>"bar","--count"=>"0"],["foo","baz"]
> catch invalid_option $ getopt opts ["-h","-v"];
invalid_option "-v"
```

```
> getopt opts [foo, "-h", bar];
["--help"=>()],[foo,bar]
```

As the last example shows, non-option arguments (as well as option values specified as separate arguments) can actually be any values which are just copied to the result lists as is.

Chapter 3

pure-doc

Version 0.6, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

pure-doc is a simple utility for literate programming and documenting source code written in the Pure programming language. It is designed to be used with the excellent docutils tools and the gentle markup format supported by these, called RST a.k.a. "reStructuredText", usually pronounced "rest".

The basic idea is that you just comment your code as usual, but using RST markup instead of plain text. In addition, you can also designate literate programming fragments in your code, which will be translated to RST literal blocks automatically. You then run pure-doc on your source files to extract all marked up comments and the literate code blocks. The resulting RST source can then be processed with the docutils utilities like rst2html.py and rst2latex.py to create the documentation in a variety of formats.

3.1 Copying

Copyright (c) 2009-2010 by Albert Graef.

pure-doc is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

3.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-doc-0.6.tar.gz.

Unpack and do the customary `make && sudo make install`. This only needs flex and a standards-compliant C++ compiler.

3.3 Usage

First, see the description of the RST format. RST is a very simple markup format, almost like plain text (in fact, you're looking at RST right now, this document is written in it!). You can learn enough of it to start marking up your source in about five minutes.

Second, you'll have to mark up your source comments. pure-doc recognizes comments in RST format by looking at the first non-empty line of the comment. A comment (either `/* ... */` or a contiguous sequence of `//` line comments) is assumed to contain RST format if the first non-empty line starts with `:`, `..` or `__`. Other comments are taken to be plain text and are ignored by pure-doc.

Notes:

- pure-doc makes no other assumption about the contents of marked up comments, so you can include whatever you want: titles, section headers, fields, admonitions, plain text, whatever. Just make sure that the comment starts with one of the special tokens listed above. (You can always put just `..` at the beginning of the comment to force it to be recognized, this will be treated as a comment by the docutils tools.)
- Also, pure-doc makes very few assumptions about the source; in fact, any source files with a C/C++-like comment and string syntax should work. So you could also use it to document your C/C++ programs, or even plain text files like this one, as long as they adhere to these standards.
- Indentation in extracted comments is preserved (assuming tabs = 8 spaces by default, you can change this with the `-t` option). This is important because indentation conveys document structure in RST.

For instance, here is a sample RST-formatted comment:

```
/* :Name: ''rand'' - compute random numbers
   :Synopsis: ''rand''
   :Description: Computes a (pseudo) random number. Takes no parameters.
   :Example: Here is how you can call ''rand'' in Pure:
     ::

       > extern int rand();
       > rand;
       1804289383

   :See Also: rand(3) */
```

This will be rendered as follows:

Name `rand` - compute random numbers

Synopsis `rand`

Description Computes a (pseudo) random number. Takes no parameters.

Example Here is how you can call `rand` in Pure:

```
> extern int rand();
> rand;
1804289383
```

See Also rand(3)

Finally, to extract the documentation you run pure-doc on your source files as follows:

```
pure-doc source-files ...
```

If no input files are specfied then the source is read from standard input. Otherwise all input files are read and processed in the indicated order. The output is written to stdout, so that you can directly pipe it into one of the docutils programs:

```
pure-doc source-files ... | rst2html.py
```

If you prefer to write the output to a file, you can do that as follows:

```
pure-doc source-files ... > rst-file
```

pure-doc also understands the following options. These must come before any file arguments.

`-h` Print a short help message.

`-i` Automatic index creation (see below).

`-s` Generate Sphinx-compatible output (see below).

`-twidth` Set the tab width to the given number of spaces.

There are no other options. By its design pure-doc is just a plain simple "docstring scraping" utility with no formatting knowledge of its own. All actual formatting is handled by the docutils programs which offer plenty of options to change the appearance of the generated output; please refer to the docutils documentation for details.

Note that since Pure 0.46, all Pure documentation is usually formatted using Sphinx, the RST formatter used by the Python project which provides cross-document indexing and referencing, and even more elaborate formatting options and prettier output than docutils. pure-doc versions since 0.6 support this by adding the `-s` option which makes its output compatible with Sphinx. (At present this option actually has any effect only when combined with the `-i` index generation option, see Hyperlink Targets and Index Generation below.)

3.4 Literate Programming

pure-doc also recognizes literate code delimited by comments which, besides the comment delimiters and whitespace, contain nothing but the special start and end "tags" >>> and <<<. Code between these delimiters (including all comments) is extracted from the source and output as a RST literal code block.

For instance:

```
/* ..

   pure-doc supports literate programming, too. */

// >>>

// This is a literate comment.
/* .. This too! */

extern int rand();
rand;

// <<<
```

This will be rendered as follows:

pure-doc supports literate programming, too.

```
// This is a literate comment.
/* .. This too! */

extern int rand();
rand;
```

Try it now! You can scrape all the sample "documentation" from this file and format it as html, as follows:

```
pure-doc README | rst2html.py --no-doc-title --no-doc-info > test.html
```

3.5 Hyperlink Targets and Index Generation

Note: This feature is now largely obsolete as Pure uses Sphinx for formatting its documentation these days. Thus, as of version 0.6, the indexing feature must be enabled explicitly with the `-i` option.

When run with the `-i` option, pure-doc supplements the normal hyperlink target processing by the docutils tools, by recognizing explicit hyperlink targets of the form `.. _target:` and automatically creating raw html targets (`<a name=...>`) for them. This works around the docutils name mangling (which is undesirable if you're indexing, say, function names).

It also resolves a quirk with some w3m versions which don't pick up all `id` attributes in the docutils-generated html source.

In addition, you can also have pure-doc generate an index from all explicit targets. To these ends, just add the following special directive at the place where you want the index to appear:

```
.. makeindex::
```

The directive will be replaced with a list of references to all targets collected *up to that point*, sorted alphabetically. This also resets the list of collected targets, so that you can have multiple smaller indices in your document instead of one big one.

It goes without saying that this facility is rather simplistic, but it may be useful when you are working with plain docutils which does not provide its own indexing facility. Note, however, that docutils doesn't allow multiple explicit targets with the same name, so you should take that into consideration when devising your index terms.

Also note that in Sphinx compatibility mode (`-s`), pure-doc will generate the appropriate Sphinx markup for index entries (`index::`) instead, and the `makeindex::` directive will be ignored. You should then use Sphinx to generate the index.

Finally, if the `-i` option isn't specified, then all this special processing is disabled and the `makeindex::` directive won't be recognized at all. This is the recommended way to process Pure documentation files which have been fully converted to Sphinx.

3.6 Generating and Installing Local Documentation

Note: This section only applies to 3rd party packages with their own bundled documentation which isn't part of the "official" Pure documentation. In this case it is possible to use docutils or some other RST formatting software to generate additional documentation files for use with the Pure interpreter. Please note that the method sketched out in this section doesn't provide full integration with the rest of Pure's documentation, but at least it makes it possible to read the local documentation in the interpreter.

If you're generating some library documentation for which you have to process a bigger collection of source files, then it is often convenient to have a few Makefile rules to automatize the process. To these ends, simply add rules similar to the following to your Makefile (the following assumes GNU make and that you're using docutils to format the documentation):

```
# The sources. Order matters here. The generated documentation will have the
# comments from each source file in the indicated order.
sources = foo.pure bar.pure

# The basename of the documentation files to be generated.
target = foo

.PHONY: html tex pdf
```

```
html: $(target).html
tex: $(target).tex
pdf: $(target).pdf

$(target).txt: $(sources)
        pure-doc $(sources) > $@

# This requires that you have docutils installed.

%.html: %.txt
        rst2html.py $< $@

%.tex: %.txt
        rst2latex.py $< $@

# This also requires that you have TeX installed.

%.pdf: %.tex
        pdflatex $<
        rm -f *.aux *.log *.out

clean:
        rm -f *.html *.tex *.pdf
```

You might want to add `-i` to the pure-doc command line if you want to enable the indexing feature described in the previous section. If you want to use some other RST formatting software, please check the corresponding documentation for information on how to format your documents and adjust the above rules for the html, tex and pdf targets accordingly.

Now you can just type `make html` to generate the documentation in html format, and `make tex` or `make pdf` to generate the other formats. The `clean` target removes the generated files.

Having generated the documentation files in html format, you can install them in the docs subdirectory of the Pure library directory to make it known to the Pure interpreter, so that you can read your documentation with the `help` command of the interpreter. (When doing this, name your documentation files in such a manner that you don't overwrite any of the Pure documentation files there.) The following Makefile rule automatizes this process. Add this to the Makefile in the previous section:

```
# Try to guess the installation prefix (this needs GNU make):
prefix = $(patsubst %/bin/pure,%,$(shell which pure 2>/dev/null))
ifeq ($(strip $(prefix)),)
# Fall back to /usr/local.
prefix = /usr/local
endif

libdir = $(prefix)/lib
docsdir = $(libdir)/pure/docs

install:
        test -d "$(DESTDIR)$(docsdir)" || mkdir -p "$(DESTDIR)$(docsdir)"
```

```
        cp $(target).html "$(DESTDIR)$(docsdir)"
```

After a `make install` your documentation should now end up in the appropriate place in the Pure library directory and you can read it in the Pure interpreter using a command like the following:

```
> help foo#
```

Note the hash character. This tells the `help` command that this is an auxiliary documentation file, rather than a search term to be looked up in the Pure documentation. You can also look up a specific section in your manual as follows:

```
> help foo#section-name
```

Please also refer to *The Pure Manual* for more information on how to use the interpreter's online help.

3.7 Formatting Tips

If you're generating documentation in pdf format using plain docutils, you might have to fiddle with the formatting to get results suitable for publication purposes. Newer versions of the rts2latex.py program provide some options which let you adjust the formatting of various document elements. Here are the options that the author found particularly helpful:

- The table of contents that RST produces isn't all that useful in printed documentation, since it lacks page numbers. As a remedy, you can invoke rst2latex with `--use-latex-toc` to have LaTeX handle the formatting of the table of contents, which looks much nicer.
- Similarly, `--use-latex-docinfo` can be used to tell rst2latex that you want the title information (author and date) to be formatted the LaTeX way.
- If you need specific LaTeX document options, these can be specified with `--documentoptions`, e.g.: `--documentoptions="11pt"`.
- For more comprehensive formatting changes which require special LaTeX code and/or packages, you can use the `--stylesheet` option. E.g., `--stylesheet=preamble.tex` will cause a preamble.tex file with your own definitions to be included in the preamble of the generated document.
- To format literal code blocks using an alternative environment instead of the default verbatim environment, use the `--literal-block-env` option. E.g., `--literal-block-env=lstlisting` will use the highlighted code environment from the listings package. (Note that in this case you'll also need a preamble which loads the corresponding package.).

To learn more about this, please consult the rts2latex.py documentation at the docutils website.

In addition, the pure-doc package contains a little GNU awk script called fixdoc, which attempts to improve the LaTeX output produced by older svn versions of rst2latex in various ways. (This isn't necessary for the latest rst2latex versions, or if you use Sphinx.)

Chapter 4

pure-ffi

Version 0.13, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

The libffi library provides a portable, high level programming interface to various calling conventions. This allows a programmer to call any function specified by a call interface description at run time. libffi should be present on most gcc-based systems, but it is also available as a standalone package at http://sourceware.org/libffi/.

This module provides an interface to libffi which enables you to call C functions from Pure and vice versa. It goes beyond Pure's built-in C interface in that it also handles C structs and makes Pure functions callable from C. Moreover, depending on the libffi implementation, it may also be possible to call foreign languages other than C.

4.1 Copying

Copyright (c) 2008, 2009 by Albert Graef.

pure-ffi is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

4.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-ffi-0.13.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure and libffi installed.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. Please see the Makefile for details.

NOTE: This module requires libffi 3.x (3.0.8 has been tested). Old libffi versions (2.x) do not appear to work (closures are broken). Patches are welcome.

4.3 Usage

The module exposes a simplified interface to libffi tailored to the Pure language. Call interfaces are described using the desired ABI, return type and tuple of argument types. The ABI is specified using one of the `FFI_*` constants defined by the module; for most purposes, `FFI_DEFAULT_ABI` is all that's needed. C types are specified using special descriptors `void_t`, `uint_t` etc., see ffi.pure for details. You can also get a list of these values using `show -g FFI_* *_t` after importing the ffi module.

The primary interface for calling C from Pure and vice versa is as follows:

`fcall` name abi rtype atypes
: Creates a Pure function from a C function with the given name, specified as a string. This makes the C function callable in Pure, no matter whether it is already declared as an `extern` or not. But note that if the function resides in a shared library, you still have to import that library using a Pure `using` declaration, see the Pure manual for details.

`fclos` fn abi rtype atypes
: Creates a pointer to a C function from the given Pure function `fn`. The resulting pointer can then be passed to other C functions expecting functions as arguments. This allows you to create C callbacks from Pure functions without writing a single line of C code. (This functionality might not be available on some platforms.)

Note that in difference to `extern` functions, arguments to functions created with libffi are always passed in uncurried form, as a Pure tuple. E.g.:

```
> using ffi;
> let fmod = fcall "fmod" FFI_DEFAULT_ABI double_t (double_t,double_t);
> fmod (5.3,0.7);
0.4
```

C structs are fully supported and are passed in a type-safe manner, see ffi.pure for details. Note that these are to be used for passing structs by value. (When passing a pointer to a struct, you must use `pointer_t` instead.) For instance:

```
> let complex_t = struct_t (double_t,double_t);
> let cexp = fcall "cexp" FFI_DEFAULT_ABI complex_t (complex_t);
> members (cexp (struct complex_t (0.0,1.0)));
0.54030230586814,0.841470984807897
```

See the examples folder in the sources for more examples.

4.4 TODO

The API isn't perfect yet. In particular, one might consider to implement type descriptors as structs instead of raw pointers, and support for typed pointers would be useful. Contributions and suggestions are welcome.

Chapter 5

pure-gen: Pure interface generator

Version 0.15, June 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

pure-gen is a C interface generator for the Pure language. It takes a C header file as input and generates a corresponding Pure module with the constant definitions and extern declarations needed to use the C module from Pure. pure-gen can also generate FFI interfaces rather than externs (using the *pure-ffi* module), and it can optionally create a C wrapper module which allows you to create interfaces to pretty much any code which can be called via C.

5.1 Synopsis

```
pure-gen [options ...] input-file
```

5.2 Options

5.2.1 General Options

-h
--help
: Print a brief help message and exit.

-V
--version
: Print version number and exit.

-e

--echo
Echo preprocessor lines. Prints all processed `#defines`, useful for debugging purposes.

-v
--verbose
Show parameters and progress information. Gives useful information about the conversion process.

-w[level]
--warnings[=level]
Display warnings, `level` = 0 (disable most warnings), 1 (default, shows important warnings only) or 2 (lots of additional warnings useful for debugging purposes).

5.2.2 Preprocessor Options

-I path
--include path
Add include path. Passed to the C preprocessor.

-D name[=value]
--define name[=value]
Define symbol. Passed to the C preprocessor.

-U name
--undefine name
Undefine symbol. Passed to the C preprocessor.

-C option
--cpp option
Pass through other preprocessor options and arguments.

5.2.3 Generator Options

-f iface
--interface iface
Interface type (`extern`, `c`, `ffi` or `c-ffi`). Default is `extern`. The `extern` and `c` types generate Pure `extern` declarations, which is what you want in most cases. `ffi` and `c-ffi` employ Pure's libffi interface instead. The `c` and `c-ffi` types cause an additional C wrapper module to be created (see Generating C Code). These can also be combined with the `-auto` suffix which creates C wrappers only when needed to get C struct arguments and returns working, see Dealing with C Structs for details.

-l lib
--lib-name lib
Add dynamic library module to be imported in the Pure output file. Default is `-l c-file` (the filename specified with *-c*, see below, without filename extension) if one of the `-fc` options was specified, none otherwise.

-m name
--namespace name
: Module namespace in which symbols should be declared.

-p prefix
--prefix prefix
: Module name prefix to be removed from C symbols.

-P prefix
--wrap prefix
: Prefix to be prepended to C wrapper symbols (`-fc` and friends). Default is `Pure_`.

-a
--all
: Include "hidden" symbols in the output. Built-in preprocessor symbols and symbols starting with an underscore are excluded unless this option is specified.

-s pattern
--select pattern
: Selection of C symbols to be included in the output. `pattern` takes the form `[glob-patterns::][regex-pattern]`, designating a comma separated list of glob patterns matching the source filenames, and an extended regular expression matching the symbols to be processed. See glob(7) and regex(7). The default `pattern` is empty which matches all symbols in all source modules.

-x pattern
--exclude pattern
: Like *-s*, but *excludes* all matching C symbols from the selection.

-t file
--template file
: Specify a C template file to be used with C wrapper generation (`-fc`). See Generating C Code for details.

-T file
--alt-template file
: Specify an alternate C template file to be used with C wrapper generation (`-fc`). See Generating C Code for details.

5.2.4 Output Options

-n
--dry-run
: Only parse without generating any output.

-N
--noclobber
: Append output to existing files.

-o file

--output file
: Pure output (.pure) filename. Default is `input-file` with new extension .pure.

-c file
--c-output file
: C wrapper (.c) filename (`-fc`). Default is `input-file` with new extension .c.

5.3 Description

pure-gen generates Pure bindings for C functions from a C header file. For instance, the command

```
pure-gen foo.h
```

creates a Pure module foo.pure with `extern` declarations for the constants (`#defines` and enums) and C routines declared in the given C header file and (recursively) its includes.

pure-gen only accepts a single header file on the command line. If you need to parse more than one header in a single run, you can just create a dummy header with all the necessary `#includes` in it and pass that to pure-gen instead.

When invoked with the *-n* option, pure-gen performs a dry run in which it only parses the input without actually generating any output files. This is useful for checking the input (possibly in combination with the *-e*, *-v* and/or *-w* options) before generating output. A particularly useful example is

```
pure-gen -ne foo.h \
  | awk '$1=="#" && $2~/^[0-9]+$/ && $3!~/^"<.*>"$/  { print $3 }' \
  | sort | uniq
```

which prints on standard output all headers which are included in the source. This helps to decide which headers you want to be included in the output, so that you can set up a corresponding filter patterns (*-s* and *-x* options, see below).

The *-I*, *-D* and *-U* options are simply passed to the C preprocessor, as well as any other option or argument escaped with the *-C* flag. This is handy if you need to define additional preprocessor symbols, add directories to the include search path, etc., see cpp(1) for details.

There are some other options which affect the generated output. In particular, `-f c` generates a C wrapper module along with the Pure module (see Generating C Code below), and `-f ffi` generates a wrapper using Pure's ffi module. Moreover, `-l libfoo` generates a `using "lib:libfoo"` declaration in the Pure source, for modules which require a shared library to be loaded. Any number of *-l* options can be specified.

Other options for more advanced uses are explained in the following sections.

5.4 Filtering

Note that pure-gen always parses the given header file as well as *all* its includes. If the header file includes system headers, by default you will get those declarations as well. This is often undesirable. As a remedy, pure-gen normally excludes built-in `#define`s of the C preprocessor, as well as identifiers with a leading underscore (which are often found in system headers) from processing. You can use the *-a* option to disable this, so that all these symbols are included as well.

In addition, the *-s* and *-x* options enable you to filter C symbols using the source filename and the symbol as search criteria. For instance, to just generate code for a single header foo.h and none of the other headers included in foo.h, you can invoke pure-gen as follows:

```
pure-gen -s foo.h:: foo.h
```

Note that even in this case all included headers will be parsed so that `#defined` constants and enum values can be resolved, but the generated output will only contain definitions and declarations from the given header file.

In general, the *-s* option takes an argument of the form `glob-patterns::regex-pattern` denoting a comma-separated list of glob patterns to be matched against the source filename in which the symbol resides, and an extended regex to be matched against the symbol itself. The `glob-patterns::` part can also be omitted in which case it defaults to `::` which matches any source file. The regex can also be empty, in which case it matches any symbol. The generated output will contain only the constant and function symbols matching the given regex, from source files matching any of the the glob patterns. Thus, for instance, the option `-s foo.h,bar.h::^(foo|bar)_` pulls all symbols prefixed with either `foo_` or `bar_` from the files foo.h and bar.h in the current directory.

Instead of `::` you can also use a single semicolon `;` to separate glob and regex pattern. This is mainly for Windows compatibility, where the msys shell sometimes eats the colons or changes them to `;`.

The *-x* option works exactly the same, but *excludes* all matching symbols from the selection. Thus, e.g., the option `-x ^bar_` causes all symbols with the prefix `bar_` to *not* be included in the output module.

Processing of glob patterns is performed using the customary rules for filename matching, see glob(7) for details. Note that some include files may be specified using a full pathname. This is the case, in particular, for system includes such as `#include <stdio.h>`, which are resolved by the C preprocessor employing a search of the system include directories (as well as any directories named with the *-I* option).

Since the `*` and `?` wildcards never match the pathname separator `/`, you have to specify the path in the glob patterns in such cases. Thus, e.g., if the foo.h file actually lives in either /usr/include or /usr/local/include, then it must be matched using a pattern like `/usr/include/*.h,/usr/local/include/*.h::`. Just `foo.h::` will not work in this case. On the other hand, if you have set up your C sources in some local directory then specifying a relative pathname is ok.

5.5 Name Mangling

The *-s* option is often used in conjuction with the *-p* option, which lets you specify a "module name prefix" which should be stripped off from C symbols. Case is insignificant and a trailing underscore will be removed as well, so `-p foo` turns `fooBar` into `Bar` and `FOO_BAR` into `BAR`. Moreover, the *-m* option allows you to specify the name of a Pure namespace in which the resulting constants and functions are to be declared. So, for instance, `-s "^(foo|FOO)" -p foo -m foo` will select all symbols starting with the `foo` or `FOO` prefix, stripping the prefix from the selected symbols and finally adding a `foo::` namespace qualifier to them instead.

5.6 Generating C Code

As already mentioned, pure-gen can be invoked with the `-fc` or `-fc-ffi` option to create a C wrapper module along with the Pure module it generates. There are various situations in which this is preferable, e.g.:

- You are about to create a new module for which you want to generate some boilerplate code.
- The C routines to be wrapped aren't available in a shared library, but in some other form (e.g., object file or static library).
- You need to inject some custom code into the wrapper functions (e.g., to implement custom argument preprocessing or lazy dynamic loading of functions from a shared library).
- The C routines can't be called directly through Pure externs.

The latter case might arise, e.g., if the module uses non-C linkage or calling conventions, or if some of the operations to be wrapped are actually implemented as C macros. (Note that in order to wrap macros as functions you'll have to create a staged header which declares the macros as C functions, so that they are wrapped in the C module. pure-gen doesn't do this automatically.)

Another important case is that some of the C routines pass C structs by value or return them as results. This is discussed in more detail in the following section.

For instance, let's say that we want to generate a wrapper foo.c from the foo.h header file whose operations are implemented in some library libfoo.a or libfoo.so. A command like the following generates both the C wrapper and the corresponding Pure module:

```
pure-gen -fc foo.h
```

This creates foo.pure and foo.c, with an import clause for `"lib:foo"` at the beginning of the Pure module. (You can also change the name of the Pure and C output files using the *-o* and *-c* options, respectively.)

The generated wrapper is just an ordinary C file which should be compiled to a shared object (dll on Windows) as usual. E.g., using gcc on Linux:

```
gcc -shared -o foo.so foo.c -lfoo
```

That's all. You should now be able to use the foo module by just putting the declaration `using foo;` into your programs. The same approach also works with the ffi interface if you replace the `-fc` option with `-fc-ffi`.

You can also adjust the C wrapper code to some extent by providing your own template file, which has the following format:

```
/* frontmatter here */
#include %h
%%

/* wrapper here */
%r %w(%p)
{
  return %n(%a);
}
```

Note that the code up to the symbol `%%` on a line by itself denotes "frontmatter" which gets inserted at the beginning of the C file. (The frontmatter section can also be empty or missing altogether if you don't need it, but usually it will contain at least an `#include` for the input header file.)

The rest of the template is the code for each wrapper function. Substitutions of various syntactical fragments of the function definition is performed using the following placeholders:

`%h` input header file

`%r` return type of the function

`%w` the name of the wrapper function

`%p` declaration of the formal parameters of the wrapper function

`%n` the real function name (i.e., the name of the target C function to be called)

`%a` the arguments of the function call (formal parameters with types stripped off)

`%%` escapes a literal %

A default template is provided if you don't specify one (which looks pretty much like the template above, minus the comments). A custom template is specified with the *`-t`* option. (There's also a *`-T`* option to specify an "alternate" template for dealing with routines returning struct values, see Dealing with C Structs.)

For instance, suppose that we place the sample template above into a file foo.templ and invoke pure-gen on the foo.h header file as follows:

```
pure-gen -fc -t foo.templ foo.h
```

Then in foo.c you'd get C output code like the following:

```
/* frontmatter here */
#include "foo.h"

/* wrapper here */
void Pure_foo(int arg0, void* arg1)
{
  return foo(arg0, arg1);
}

/* wrapper here */
int Pure_bar(int arg0)
{
  return foo(arg0);
}
```

As indicated, the wrapper function names are usually stropped with the `Pure_` prefix. You can change this with the *-P* option.

This also works great to create boilerplate code for new modules. For this purpose the following template will do the trick:

```
/* Add #includes etc. here. */
%%

%r %n(%p)
{
  /* Enter code of %n here. */
}
```

5.7 Dealing with C Structs

Modern C compilers allow you to pass C structs by value or return them as results from a C function. This represents a problem, because Pure doesn't provide any support for that in its extern declarations. Even Pure's libffi interface only has limited support for C structs (no unions, no bit fields), and at present pure-gen itself does not keep track of the internal structure of C structs either.

Hence pure-gen will bark if you try to wrap an operation which passes or returns a C struct, printing a warning message like the following which indicates that the given function could not be wrapped:

```
Warning: foo: struct argument or return type, try -fc-auto
```

What Pure *does* know is how to pass and return *pointers* to C structs in its C interface. This makes it possible to deal with struct arguments and return values in the C wrapper. To make this work, you need to create a C wrapper module as explained in the previous section. However, as C wrappers are only needed for functions which actually have struct arguments or return values, you can also use the `-fc-auto` option (or `-fc-ffi-auto` if you prefer the

ffi interface) to only generate the C wrapper when required. This saves the overhead of an extra function call if it's not actually needed.

Struct arguments in the original C function then become struct pointers in the wrapper function. E.g., if the function is declared in the header as follows:

```
typedef struct { double x, y; } point;
extern double foo(point p);
```

Then the generated wrapper code becomes:

```
double Pure_foo(point* arg0)
{
  return foo(*arg0);
}
```

Which is declared in the Pure interface as:

```
extern double Pure_foo(point*) = foo;
```

Struct return values are handled by returning a pointer to a static variable holding the return value. E.g.,

```
extern point bar(double x, double y);
```

becomes:

```
point* Pure_bar(double arg0, double arg1)
{
  static point ret;
  ret = bar(arg0, arg1); return &ret;
}
```

Which is declared in the Pure interface as:

```
extern point* Pure_bar(double, double) = bar;
```

(Note that the generated code in this case comes from an alternate template. It's possible to configure the alternate template just like the normal one, using the `-T` option instead of `-t`. See the Generating C Code section above for details about code templates.)

In a Pure script you can now call `foo` and `bar` as:

```
> foo (bar 0.0 1.0);
```

Note, however, that the pointer returned by `bar` points to static storage which will be overwritten each time you invoke the `bar` function. Thus in the following example *both* `u` and `v` will point to the same `point` struct, namely that defined by the latter call to `bar`:

```
> let u = bar 1.0 0.0; let v = bar 0.0 1.0;
```

Which most likely is *not* what you want. To avoid this, you'll have to take dynamic copies of returned structs. It's possible to do this manually by fiddling around with `malloc` and

memcpy, but the most convenient way is to employ the struct functions provided by Pure's ffi module:

```
> using ffi;
> let point_t = struct_t (double_t, double_t);
> let u = copy_struct point_t (bar 1.0 0.0);
> let v = copy_struct point_t (bar 0.0 1.0);
```

Now u and v point to different, malloc'd structs which even take care of freeing themselves when they are no longer needed. Moreover, the ffi module also allows you to access the members of the structs in a direct fashion. Please refer to the *pure-ffi* documentation for further details.

5.8 Notes

pure-gen currently requires gcc (-E) as the C preprocessor. It also needs a version of gcc which understands the -fdirectives-only option, which means gcc 4.3 or later. It will run with older versions of gcc, but then you'll get an error message from gcc indicating that it doesn't understand the -fdirectives-only option. pure-gen then won't be able to extract any #defined constants from the header files.

pure-gen itself is written in Pure, but uses a C parser implemented in Haskell, based on the Language.C library written by Manuel Chakravarty and others.

pure-gen can only generate C bindings at this time. Other languages may have their own calling conventions which make it hard or even impossible to call them directly through Pure's extern interface. However, if your C compiler knows how to call the other language, then it may be possible to interface to modules written in that language by faking a C header for the module and generating a C wrapper with a custom code template, as described in Generating C Code. In principle, this approach should even work with behemoths like C++, although it might be easier to use third-party tools like SWIG for that purpose.

In difference to SWIG and similar tools, pure-gen doesn't require you to write any special "interface files", is controlled entirely by command line options, and the amount of marshalling overhead in C wrappers is negligible. This is possible since pure-gen targets only the Pure-C interface and Pure has good support for interfacing to C built into the language already.

pure-gen usually works pretty well if the processed header files are written in a fairly clean fashion. Nevertheless, some libraries defy fully automatic wrapper generation and may thus require staged headers and/or manual editing of the generated output to get a nice wrapper module.

In complex cases it may also be necessary to assemble the output of several runs of pure-gen for different combinations of header files, symbol selections and/or namespace/prefix settings. In such a situation it is usually possible to just concatenate the various output files produced by pure-gen to consolidate them into a single wrapper module. To make this easier, pure-gen provides the *-N* a.k.a. *--noclobber* option which appends the output to existing files instead of overwriting them. See the example below.

5.9 Example

For the sake of a substantial, real-world example, here is how you can wrap the entire GNU Scientific Library in a single Pure module mygsl.pure, with the accompanying C module in mygsl.c:

```
rm -f mygsl.pure mygsl.c
DEFS=-DGSL_DISABLE_DEPRECATED
for x in /usr/include/gsl/gsl_*.h; do
  pure-gen $DEFS -N -fc-auto -s "$x::" $x -o mygsl.pure -c mygsl.c
done
```

The C module can then be compiled with:

```
gcc $DEFS -shared -o mygsl.so mygsl.c
```

Note that the `GSL_DISABLE_DEPRECATED` symbol must be defined here to avoid some botches with constants being defined in incompatible ways in different GSL headers. Also, some GSL versions have broken headers lacking some system includes which causes hiccups in pure-gen's C parser. Fixing those errors or working around them through some appropriate cpp options should be a piece of cake, though.

5.10 License

BSD-like. See the accompanying COPYING file for details.

5.11 Authors

Scott E. Dillard (University of California at Davis), Albert Graef (Johannes Gutenberg University at Mainz, Germany).

5.12 See Also

Language.C A C parser written in Haskell by Manuel Chakravarty et al, http://www.sivity.net/projects/language.c.

SWIG The Simplified Wrapper and Interface Generator, http://www.swig.org.

Chapter 6

pure-readline

Version 0.1, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

Get the latest source from http://pure-lang.googlecode.com/files/pure-readline-0.1.tar.gz.

This is a trivial wrapper around GNU readline, which gives Pure scripts access to the `readline` and `add_history` functions. The wrapper can also be used with the BSD editline a.k.a. libedit library, a readline replacement licensed under the 3-clause BSD license. You can find these at:

- GNU readline: http://tiswww.tis.case.edu/~chet/readline/rltop.html
- BSD editline/libedit: http://www.thrysoee.dk/editline

We recommend GNU readline because it's easier to use and has full UTF-8 support, but in some situations BSD editline/libedit may be preferable for license reasons or because it's what the operating system provides. Note that in either case Pure programs using this module are subject to the license terms of the library that you use (GPLv3+ in case of GNU readline, BSD license in the case of BSD editline/libedit).

Normally, you should choose the same library that you use with the Pure interpreter, to avoid having two different versions of the library linked into your program. (This doesn't matter if you only use this module with batch-compiled scripts, though, since the Pure runtime doesn't depend on readline in any way.) By default, the module will be built with GNU readline. To select editline/libedit instead, you only have to uncomment a line at the beginning of the Makefile. Also, you might want to check the beginning of readline.c for the proper location of the corresponding header files.

The module provides two functions:

`readline` prompt
: Read a line of input from the user, with prompting and command line editing. Returns the input line (with the trailing newline removed), or `NULL` when reaching end of file.

add_history line

Adds the given line (a string) to the command history.

Example:

```
> readline "input> ";
input> Hello, world!
"Hello, world!"
> add_history ans;
()
> readline "input> ";
input> <EOF>
#<pointer 0>
```

Chapter 7

pure-sockets: Pure Sockets Interface

Version 0.6, June 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

This is an interface to the Berkeley socket functions. It provides most of the core functionality, so you can create sockets for both stream and datagram based protocols and use these to transmit messages. Unix-style file sockets are also available if the host system supports them.

7.1 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-sockets-0.6.tar.gz.

Run `make` to compile the module and `sudo make install` to install it in the Pure library directory. To uninstall the module, use `sudo make uninstall`. There are a number of other targets (mostly for maintainers), please see the Makefile for details.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. You can also set custom compilation options with the CFLAGS variable, e.g.: `make CFLAGS=-O3`. Again, please see the Makefile for details.

7.2 Usage

To use the operations of this module, put the following in your Pure script:

```
using sockets;
```

With the sockets module loaded, all the standard socket functions are available and work pretty much like in C. The only real difference is that, for convenience, functions taking socket addresses as parameters (struct_sockaddr* pointers in Pure), are called without the addrlen parameter; the size of the socket address structure will be inferred automatically and passed to the underlying C functions. Also, there are some convenience functions which act as wrappers around getaddrinfo and getnameinfo to create socket addresses from symbolic information (hostname or ip, port names or numbers) and return information about existing address pointers, see Creating and Inspecting Socket Addresses below.

Below is a list of the provided functions. Please see the corresponding manual pages for details, and check the Pure scripts in the examples subdirectory for some examples.

7.2.1 Creating and Inspecting Socket Addresses

These functions are Pure-specific. The created socket addresses are malloc'ed and free themselves automatically when garbage-collected.

sockaddr ()
: Create a pointer to an empty socket address suitable to hold the socket address result of routines like accept, getsockname, recvfrom, etc. which return a socket address.

sockaddr ([int family,] char *path)
: Create a local (a.k.a. file) socket address for the given pathname. The family parameter, if specified, must be AF_UNIX here. Please note that AF_UNIX is not supported on all platforms. You can check for this by testing the HAVE_AF_UNIX constant, which is a truth value specifying whether AF_UNIX is available on your system.

sockaddr ([int family,] char *host, char *port)
sockaddr ([int family,] char *host, int port)
: This uses getaddrinfo to retrieve an AF_INET or AF_INET6 address for the given hostname (or numeric IP address in string form) and port (specified either as an int or a string). If family is omitted, it defaults to AF_UNSPEC which matches both AF_INET and AF_INET6 addresses.

sockaddrs ([int family,] char *host, char *port)
sockaddrs ([int family,] char *host, int port)
: This works like sockaddr above, but returns a list with *all* matching addresses.

sockaddr_family addr
: Returns the address family of the given address.

sockaddr_path addr
: Returns the pathname for AF_UNIX addresses.

sockaddr_hostname addr
: Returns the hostname if available, the IP address otherwise.

sockaddr_ip addr
: Returns the IP address.

sockaddr_service addr
Returns the service (a.k.a. port) name.

sockaddr_port addr
Returns the port number.

sockaddr_info addr
Returns a readable description of a socket address, as a `(family,hostname,port)` tuple. You should be able to pass this into `sockaddr` again to get the original address.

7.2.2 Creating and Closing Sockets

socket domain type protocol
Creates a socket for the given protocol family (`AF_UNIX`, `AF_INET` or `AF_INET6`), socket type (`SOCK_STREAM`, `SOCK_DGRAM`, etc.) and protocol. Note that on Linux we also support the `SOCK_NONBLOCK` (non-blocking) and `SOCK_CLOEXEC` (close-on-exec) flags which can be or'ed with the socket type to get sockets with the corresponding features. The protocol number is usually 0, denoting the default protocol, but it can also be any of the prescribed `IPPROTO` constants (a few common ones are predefined by this module, try `show -g IPPROTO_*` for a list of those).

socketpair domain type protocol sv
Create a pair of sockets. The descriptors are returned in the integer vector `sv` passed in the last argument.

shutdown fd how
Perform shutdown on a socket. The second argument should be one of `SHUT_RD`, `SHUT_WR` and `SHUT_RDWR`.

closesocket fd
This is provided for Windows compatibility. On POSIX systems this is just `close`.

7.2.3 Establishing Connections

accept sockfd addr

bind sockfd addr

connect sockfd addr

listen sockfd backlog

7.2.4 Socket I/O

recv fd buf len flags

send fd buf len flags

recvfrom fd buf len flags addr

sendto fd buf len flags addr

The usual send/recv flags specified by POSIX (MSG_EOR, MSG_OOB, MSG_PEEK, MSG_WAITALL) are provided. On Linux we also support MSG_DONTWAIT. Note that on POSIX systems you can also just fdopen the socket descriptor and use the standard file I/O operations from the system module instead.

7.2.5 Socket Information

getsockname fd addr

getpeername fd addr

getsockopt fd level name val len

setsockopt fd level name val len

For getsockopt and setsockopt, currently only the SOL_SOCKET level is defined (level argument) along with the available POSIX socket options (name argument). Try show -g SO_* to get a list of those. Also note that for most socket level options the val argument is actually an int*, so you can pass a Pure int vector (with len = SIZEOF_INT) for that parameter.

7.3 Example

Here is a fairly minimal example using Unix stream sockets. To keep things simple, this does no error checking whatsoever and just keeps sending strings back and forth. More elaborate examples can be found in the examples directory in the sources.

```
using sockets, system;

const path = "server_socket";
extern int unlink(char *name);

server = loop with
  loop = loop if ~null s && ~response fp s when
    // Connect to a client.
    cfd = accept fd $ sockaddr ();
    // Open the client socket as a FILE* and read a request.
    fp = fdopen cfd "r+"; s = fgets fp;
  end;
  loop = puts "server is exiting" $$ closesocket fd $$
         unlink path $$ () otherwise;
  response fp s::string = s=="quit\n" when
    // Process the request. (Here we just print the received
    // message and echo it back to the client.)
    printf "server> %s" s;
    fputs s fp;
  end;
end when
```

```
  // Create the server socket and start listening.
  unlink path;
  fd = socket AF_UNIX SOCK_STREAM 0;
  bind fd (sockaddr path); listen fd 5;
  printf "server listening at '%s'\n" path;
end;

client = loop with
  // Keep reading requests from stdin.
  loop = loop if ~null s && ~request s when
    fputs "client> " stdout; s = fgets stdin;
  end;
  loop = puts "client is exiting" $$ () otherwise;
  request s::string = s=="quit\n" when
    fd = socket AF_UNIX SOCK_STREAM 0;
    connect fd (sockaddr path);
    // Send the request to the server.
    fp = fdopen fd "r+"; fputs s fp;
    // Get the reply.
    s = fgets fp;
  end;
end;
```

To use this example, run the `server` function in one instance of the Pure interpreter and the `client` function in another. Enter a line when the client prompts you for input; it will be printed by the server. Behind the scenes, the server also sends the line back to the client. After receiving the reply, the client prompts for the next input line. Entering end-of-file at the client prompt terminates the client but keeps the server running, so that you can start another client and reconnect to the server. Entering just `quit` in the client terminates both server and client. Here is how a typical interaction may look like:

```
> client;
client> 1+1
client> foo bar
client> quit
client is exiting
()

> server;
server listening at 'server_socket'
server> 1+1
server> foo bar
server> quit
server is exiting
()
```

Note that while the server processes requests sequentially, it accepts connections from a new client after each request, so that you can run as many clients as you like.

Chapter 8

pure-stldict

Version 0.5, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This package provides a light-weight, no frills interface to the C++ dictionary containers `map` and `unordered_map`. The `stldict` module makes these data structures available in Pure land and equips them with a (more or less) idiomatic Pure container interface.

The C++ containers are part of the standard C++ library, see the C++ standard library documentation for details. They were originally based on the Standard Template Library, so they are also sometimes referred to as "STL containers"; hence the name of this package.

8.1 Copying

Copyright (c) 2011 by Albert Graef.

pure-stldict is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

8.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-stldict-0.5.tar.gz.

Run `make` to compile the modules and `make install` (as root) to install them in the Pure library directory. This requires GNU make, and of course you need to have Pure (and a C++ library which includes the STL) installed.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually, please check the Makefile for details.

Note: This module requires Pure 0.50 or later and a recent version of the C++ library (GNU libstdc++ v3 has been tested). All proper C++11 libraries should work out of the box, while (recent) C++0x implementations may require some fiddling with the sources and/or the compilation options. Pre C++0x library versions surely require considerably more work, especially in the hashdict module.

8.3 Usage

After installation, you can use the operations of this package by placing the following import declaration in your Pure programs:

```
using stldict;
```

This imports the whole shebang. If you only need either the hashed or the ordered dictionaries, you can also import the corresponding modules separately, i.e.:

```
using hashdict;
```

or:

```
using orddict;
```

8.4 Types

In Pure land, the C++ `map` and `unordered_map` containers and their `multimap` variants are made available as a collection of four data structures:

type `hashdict`
type `hashmdict`
: Hashed (unordered) dictionary data structures. These work with arbitrary key (and value) types, like the hashed dictionary and set data structures in the standard library, and can be found in the `hashdict.pure` module.

type `orddict`
type `ordmdict`
: Ordered dictionary data structures. These require the keys to be ordered by the standard `<` predicate, like the ordered dictionary and set data structures in the standard library, and can be found in the `orddict.pure` module.

Note that `hashdict` and `hashmdict` differ in that the former has exactly one key-value association for each key in the dictionary, while the latter is a "multidict" which allows multiple values to be associated with a key. The same applies to the `orddict` and `ordmdict` types.

In addition, there are various supertypes which correspond to different unions of the hashed and ordered dictionary types. These are:

type `hashxdict`
type `ordxdict`
Denotes any kind of hashed or ordered dictionary, respectively.

type `stldict`
type `stlmdict`
Denotes any kind of singled-valued or multi-valued dictionary, respectively.

type `stlxdict`
Denotes any kind of dictionary.

For instance, you can use `hashxdict` to match both `hashdict` and `hashmdict` values. Likewise, `stlmdict` matches both `hashmdict` and `ordmdict` values. To match any kind of dictionary, use the `stlxdict` type.

These data structures are very thin wrappers around the C++ container types; in fact, they are just pointers to the C++ containers. Memory management of these objects is automatic, and customizable pretty-printing is provided as well.

All data structures offer most of the usual Pure container interface (as well as some extensions). In contrast to the standard library dictionaries, they can be used both as dictionaries (holding key => value pairs) and sets (holding only keys, without associated values), even at the same time.

The other important difference to the standard library containers is that the stldict containers are *mutable* data structures; inserting and deleting members really modifies the underlying C++ containers. (However, it is possible to take copies of the containers in situations where it's necessary to preserve value semantics.)

8.5 Operations

All types of dictionaries are simply pointers to the corresponding C++ containers which hold key-value associations where both keys and values may be arbitrary Pure expressions. The basic operations described below can be used to create, query and modify these objects. Comparisons of dictionaries are implemented as well, and the set-like operations let you combine dictionaries in different ways. These operations provide an interface similar to the usual Pure container API.

In addition, the `stldict` module provides some list-like operations on dictionaries, so that the member data can be processed and aggregated in a convenient fashion (including the ability to use dictionaries as generators in list and matrix comprehensions), and there's also an interface to C++ iterators which enables you to traverse, inspect and modify the containers in a more C++-like way. Some low-level operations are available to access information

about the underlying hash table of a hashed dictionary. Last but not least, the module also offers some operations to customize the pretty-printing of dictionary values.

When working with these data structures, please note the following special properties of this implementation:

- All dictionary types are *mutable*. Inserting and deleting members really modifies the underlying C++ data structure as a side effect of the operation. If you need value semantics, you should probably use one of the dictionary or set data structures from the standard Pure library instead. Another possibility is to take a copy of a hashdict using the `copy` function if you need to preserve the original value.
- Keys in a hashed dictionary may be stored in an apparently random order (not necessarily in the order in which they were inserted), while they are guaranteed to be in ascending order (by key) for ordered dictionaries. However, note that even in the latter case, the order of different members for the same key in a multi-valued dictionary is not specified. This must be taken into account when comparing dictionaries, see below. The order of members in a dictionary also matters when listing data from a container using, e.g., the `members`, `keys` and `vals` operations.
- Two dictionaries are considered syntactically equal iff they contain the same elements in exactly the same order, using syntactic equality on both the keys and the associated values. This test can always be done in linear time, but is of limited usefulness for most kinds of dictionaries, since the exact order of members in the dictionary may vary depending on how the dictionary was constructed. Semantic equality operations are provided which check (albeit at the cost of increased running time) whether two containers contain the same members irrespective of element order, using semantic equality on the members. Various subset comparisons are provided as well, please check the Comparisons section for details.
- Values in a dictionary can be omitted, so that a dictionary can also be used as a set data structure. This obviates the need for a separate set data structure at the cost of some (small) increase in memory usage. Also note that you can't really have a hash pair `x=>y` as a member of a set, since it always denotes a key-value association. As a remedy, you may use ordinary pairs `(x,y)` instead.

8.5.1 Basic Operations

`hashdict` xs
`hashmdict` xs
`orddict` xs
`ordmdict` xs

Create a dictionary of the corresponding type from a list, tuple or vector of its members. Members can be specified as hash pairs `x=>y` to denote a key-value association. Any other kind of value denotes a singleton key without associated value. Note that the ordered dictionaries require that the keys be ordered, i.e., the `<` predicate must be defined on them.

The same operations can also be used to construct a dictionary from another dictionary of any type. If the given dictionary is already of the corresponding type, this is a no-op (if you want to copy the dictionary instead, use the `copy` function below). Otherwise the given dictionary is converted to a new dictionary of the desired target type.

mkhashdict y xs
mkhashmdict y xs
mkorddict y xs
mkordmdict y xs
Create a dictionary from a list of keys and a constant value. The resulting dictionary has the given keys and `y` as the value for each key.

copy m
Create a new dictionary with the same type and content as `m`. This is useful if you want to preserve value semantics when using destructive update operations such as `insert` and `delete`. In such a case, `copy` can be used to take a copy of the dictionary beforehand, so that the original dictionary remains unmodified.

Note: This operation needs linear time with respect to the size of the dictionary (i.e., its number of members). If logarithmic update times are needed while still preserving value semantics, you should use the dictionary and set data structures from the standard library instead.

---

hashdictp m
hashmdictp m
orddictp m
ordmdictp m
Check whether the argument is a dictionary of the corresponding type.

hashxdictp m
ordxdictp m
stldictp m
stlmdictp m
stlxdictp m
Check whether the argument is a dictionary of the corresponding supertype.

m
The size of a dictionary (the number of members it contains).

m **!** x
Get the value stored under key `x` in the dictionary `m`. This may be `x` itself if `x` is a member of `m` but has no associated value. In the case of a multidict this actually returns a list of values (which may be empty if `m` doesn't contain `x`). Otherwise an `out_of_bounds` exception is thrown if `m` doesn't contain `x`.

null m
Test whether `m` is empty, i.e., has zero members.

member m x

Test whether `m` contains a member with key `x`.

members m
list m
: Return the list of members of `m`. The member list will be in an apparently random order in the hashed dictionary case, while it is guaranteed to be in ascending order (by key) for ordered dictionaries. The same order is also used for the other inspection operations below.

stream m
: Like `list`, but the member list is returned as a lazy list (cf. *Lazy Evaluation and Streams*) whose members will be computed on the fly as the list is being traversed; cf. Iterators.

tuple m
vector m
: Return the members as a tuple or vector.

keys m
: Return the list of keys in the dictionary.

vals m
: Return the list of corresponding values. In the case of a singleton key `x` without associated value, `x` itself is returned instead.

As already mentioned, the following modification operations are destructive, i.e., they actually modify the underlying dictionary data structure. If this is not desired, you'll first have to take a copy of the target dictionary, see `copy`.

insert m x
insert m (x=>y)
update m x y
: Insert a singleton key `x` or a key-value pair `x=>y` into `m` and return the modified dictionary. This always adds a new member in a multidict, otherwise it replaces an existing value if there is one. `update` is provided as a fully curried version of `insert`, so `update m x y` behaves exactly like `insert m (x=>y)`.

delete m x
delete m (x=>y)
: Remove the key `x` or the specific key-value pair `x=>y` from `m` (if present) and return the modified dictionary. In the multidict case, only the first member with the given key `x` or key-value pair `x=>y` is removed.

clear m
: Remove all members from `m`, making `m` an empty dictionary. Returns `()`.

8.5.2 Comparisons

The usual comparison predicates (`==`, `~=`, `<=`, `<` etc.) are defined on all dictionary types, where two dictionaries are considered "equal" (`m1==m2`) if they both contain the same `key=>value` pairs, and `m1<=m2` means that `m1` is a sub-dictionary of `m2`, i.e., all `key=>value` pairs of `m1`

are also contained in `m2` (taking into account multiplicities in the multidict case). Ordered dictionaries compare keys using equality (assuming two keys `a` and `b` to be equal if neither `a<b` nor `b<a` holds), while hashed dictionaries check for syntactical equality (using `===`). The associated values are compared using the `==` predicate if it is defined, falling back to syntactic equality otherwise.

The module also defines syntactic equality on all dictionary types, so that two dictionaries of the same type are considered syntactically equal iff they contain the same (syntactically equal) members in the same order. This is always guaranteed if two dictionaries are "identical" (the same C++ pointer), but generally the member order will depend on how the dictionary was constructed. Thus if you need to check that two dictionaries contain the same members irrespective of the order in which the members are listed, the semantic equality operation `==` should be used instead; this will also handle the case of mixed operand types.

Note that if you really need to check whether two dictionaries are the same object rather than just syntactically equal, you'll have to cast them to generic C pointers before comparing them with `===`. This can be done with the following little helper function:

```
same_dict x y = pointer_cast "void*" x === pointer_cast "void*" y;
```

8.5.3 Set-Like Operations

These operations work with mixed operand types, promoting less general types to more general ones (i.e., ordered to hashed, and single-valued to multi-valued dictionaries). The result is always a new dictionary, leaving the operands unmodified.

m1 **+** m2
: Sum: `m1+m2` adds the members of `m2` to `m1`.

m1 **-** m2
: Difference: `m1-m2` removes the members of `m2` from `m1`.

m1 `*` m2
: Intersection: `m1*m2` removes the members *not* in `m2` from `m1`.

8.5.4 List-Like Operations

The following operations are all overloaded so that they work like their list counterparts, treating their dictionary argument as if it was the member list of the dictionary:

- `do`, `map`, `catmap`, `listmap`, `rowmap`, `rowcatmap`, `colmap`, `colcatmap`
- `all`, `any`, `filter`, `foldl`, `foldl1`, `foldr`, `foldr1`, `scanl`, `scanl1`, `scanr`, `scanr1`, `sort`

Note that this includes the generic comprehension helpers `listmap`, `catmap` et al, so that dictionaries can be used as generators in list and matrix comprehensions as usual (see below for some examples).

8.5.5 Iterators

These operations give direct access to C++ iterators on dictionaries which let you query the elements and do basic manipulations of the container. The operations are available in the `stldict` namespace.

The iterator concept is somewhat alien to Pure and there are some pitfalls (most notably, destructive updates may render iterators invalid), but the operations described here are still useful in some situations, especially if you need to speed up sequential accesses to large containers or modify values stored in a container in a direct way. They are also used internally to compute lazy member lists of containers (`stream` function).

You should only use these directly if you know what you are doing. In particular, make sure to consult the C++ standard library documentation for further details on C++ iterator usage.

The following operations are provided to create an iterator for a given dictionary.

`stldict::`**`begin`** m
`stldict::`**`end`** m
: Return iterators pointing to the beginning and the end of the container. (Note that `stldict::end` *must* always be specified in qualified form since `end` is a keyword in the Pure language.)

`stldict::`**`find`** m x
: Locates a key or specific key=>value pair `x` in the container and returns an iterator pointing to the corresponding member (or `stldict::end m` if `m` doesn't contain `x`).

Note that these operations return a new iterator object for each invocation. Also, the created iterator object keeps track of the container it belongs to, so that the container isn't garbage-collected while the iterator is still being used. However, removing a member from the container (using either `delete` or `stldict::erase`) invalidates all iterators pointing to that member; the result of trying to access such an invalidated iterator is undefined (most likely your program will crash).

Similar caveats also apply to the `stream` function which, as already mentioned, uses iterators internally to implement lazy list traversal of the members of a dictionary. Thus, if you delete a member of a dictionary while traversing it using `stream`, you better make sure that this member is not the next stream element remaining to be visited; otherwise bad things will happen.

The following operations on iterators let you query and modify the contents of the underlying container:

`stldict::`**`dict`** i
: Return the dictionary to which `i` belongs.

`stldict::`**`endp`** i
: Check whether the iterator `i` points to the end of the container (i.e., past the last element).

`stldict::`**`next`** i
: Advance the iterator to the next element. Note that for convenience, in contrast to the

corresponding C++ operation this operation is non-destructive. Thus it actually creates a *new* iterator object, leaving the original iterator `i` unmodified. The operation fails if `i` is already at the end of the container.

`stldict::`**`get`** i
: Retrieve the key=>val pair stored in the member pointed to by `i` (or just the key if there is no associated value). The operation fails if `i` is at the end of the container.

`stldict::`**`put`** i y
: Change the value associated with the member pointed to by `i` to `y`, and return the new value `y`. The operation fails if `i` is at the end of the container. Note that `stldict::put` only allows you to set the associated value, *not* the key of the member.

`stldict::`**`erase`** i
: Remove the member pointed to by `i` (this invalidates `i` and all other iterators pointing to this member). The operation fails if `i` is at the end of the container.

i **==** j
i **~=** j
: Semantic equality of iterators. Two iterators are considered equal (`i == j`) if `i` and `j` point to the same element in the same container, and unequal (`i ~= j`) if they don't. (In contrast, note that iterators are in fact just pointers to a corresponding C++ data structure, and thus *syntactical* equality (`i === j`) holds only if two iterators are the same object.)

8.5.6 Low-Level Operations

The `hashdict` module also provides a few specialized low-level operations dealing with the layouts of buckets and the hash policy of the `hashdict` and `hashmdict` containers, such as `bucket_count`, `load_factor`, `rehash` etc. These operations, which are all kept in their own separate `hashdict` namespace, are useful to obtain performance-related information and modify the setup of the underlying hash table. Please check the `hashdict.pure` module and the C++ standard library documentation for further details.

8.5.7 Pretty-Printing

By default, dictionaries are pretty-printed in the format `somedict xs`, where `somedict` is the actual construction function such as `hashdict`, `orddict`, etc., and `xs` is the member list of the dictionary. This is usually convenient, as the printed expression will evaluate to an equal container when reentered as Pure code. However, it is also possible to define your own custom pretty-printing with the following function.

`hashdict_symbol` f
`hashmdict_symbol` f
`orddict_symbol` f
`ordmdict_symbol` f
: Makes the pretty-printer use the format `f xs` (where `xs` is the member list) for printing the corresponding type of dictionary.

Note that `f` may also be an operator symbol (nonfix and unary symbols work best). In the case of an outfix symbol the list brackets around the members are removed; this makes it possible to render the container in a format similar to Pure's list syntax. For instance:

```
> using stldict;
> outfix {$ $};
> orddict_symbol ({$ $});
()
> orddict (1..5);
{$1,2,3,4,5$}
```

See `orddict_examp.pure` included in the distribution for a complete example which also discusses how to make such a custom print representation reparsable.

8.6 Examples

Some basic examples showing `hashdict` in action:

```
> using stldict;
> let m = hashdict [foo=>99, bar=>bar 4711L, baz=>1..5]; m;
hashdict [foo=>99,bar=>bar 4711L,baz=>[1,2,3,4,5]]
> m!bar;
bar 4711L
> keys m;
[foo,bar,baz]
> vals m;
[99,bar 4711L,[1,2,3,4,5]]
> list m;
[foo=>99,bar=>bar 4711L,baz=>[1,2,3,4,5]]
> member m foo, member m bar;
1,1
```

Hashed multidicts (`hashmdict`):

```
> let m = hashmdict [foo=>99,baz=>1..5,baz=>bar 4711L]; m;
hashmdict [foo=>99,baz=>[1,2,3,4,5],baz=>bar 4711L]
> m!baz;
[[1,2,3,4,5],bar 4711L]
> m!foo;
[99]
```

The following example illustrates how to employ ordered dictionaries (`orddict`) as a set data structure:

```
> let m1 = orddict [5,1,3,11,3];
> let m2 = orddict (3..6);
> m1;m2;
orddict [1,3,5,11]
orddict [3,4,5,6]
> m1+m2;
orddict [1,3,4,5,6,11]
```

```
> m1-m2;
orddict [1,11]
> m1*m2;
orddict [3,5]
> m1*m2 <= m1, m1*m2 <= m2;
1,1
> m1 < m1+m2, m2 < m1+m2;
1,1
```

Of course, the same works with ordered multidicts (`ordmdict`):

```
> let m1 = ordmdict [5,1,3,11,3];
> let m2 = ordmdict (3..6);
> m1;m2;
ordmdict [1,3,3,5,11]
ordmdict [3,4,5,6]
> m1+m2;
ordmdict [1,3,3,3,4,5,5,6,11]
> m1-m2;
ordmdict [1,3,11]
> m1*m2;
ordmdict [3,5]
> m1*m2 <= m1, m1*m2 <= m2;
1,1
> m1 < m1+m2, m2 < m1+m2;
1,1
```

In fact, the binary operations (comparisons as well as the set operations +, - and *) work with any combination of dictionary operands:

```
> let m1 = hashdict (1..5);
> let m2 = ordmdict (3..7);
> m1+m2;
hashmdict [1,2,3,3,4,4,5,5,6,7]
```

Note that the operands are always promoted to the more general operand type, where hashed beats ordered and multi-valued beats single-valued dictionaries. If this is not what you want, you can also specify the desired conversions explicitly:

```
> m1+orddict m2;
hashdict [1,2,3,4,5,6,7]
> orddict m1+m2;
ordmdict [1,2,3,3,4,4,5,5,6,7]
```

Also note that the "set" operations not only work with proper sets, but also with general dictionaries:

```
> hashdict [i=>i+1|i=1..4]+hashdict [i=>i-1|i=3..5];
hashdict [1=>2,2=>3,3=>2,4=>3,5=>4]
```

All dictionary containers can be used as generators in list and matrix comprehensions:

```
> let m = hashmdict [foo=>99,baz=>1..5,baz=>bar 4711L];
> [x y | x=>y = m];
[foo 99,baz [1,2,3,4,5],baz (bar 4711L)]
> {{x;y} | x=>y = m};
{foo,baz,baz;99,[1,2,3,4,5],bar 4711L}
```

Note that in the current implementation this always computes the full member list of the dictionary as an intermediate value, which will need considerable extra memory in the case of large dictionaries. As a remedy, you can also use the `stream` function to convert the dictionary to a lazy list instead. This will often be slower, but in the case of big dictionaries the tradeoff between memory usage and execution speed might be worth considering. For instance:

```
> let m = hashdict [foo i => i | i = 1..10000];
> stream m;
(foo 1512=>1512):#<thunk 0x7fa1718350a8>
> stats -m
> #list m;
10000
0.01s, 40001 cells
> #stream m;
10000
0.1s, 16 cells
> #[y | x=>y = m; gcd y 767~=1];
925
0.05s, 61853 cells
> #[y | x=>y = stream m; gcd y 767~=1];
925
0.15s, 10979 cells
```

Chapter 9

pure-stllib

Version 0.3, June 26, 2012

Peter Summerland <p.summerland@gmail.com>

pure-stllib is an "umbrella" package that contains a pair of Pure addons, *pure-stlvec* and *pure-stlmap*. These addons provide Pure interfaces to a selection of containers provided by the C++ Standard Library, specialized to hold pointers to arbitrary Pure expressions. *pure-stlvec* is a Pure interface to C++'s vector and the STL algorithms that act on them. *pure-stlmap* is an interface to six (of the eight) of C++'s associative containers: map, set, multimap, multiset, unordered_map and unordered_set.

9.1 Copying

Copyright (c) 2011-2012 by Peter Summerland <p.summerland@gmail.com>.

All rights reserved.

pure-stllib is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

pure-stllib is distributed under a BSD-style license, see the COPYING file for details.

9.2 Installation

pure-stllib-0.3 requires at least Pure 0.50. The latest version of Pure is available at http://code.google.com/p/pure-lang/downloads/list.

The latest version of the source code for *pure-stllib* can be downloaded from http://pure-lang.googlecode.com/files/pure-stllib-0.3.tar.gz.

To install pure-stllib-0.3 (on Linux), extract the source code (e.g., tar -xzf pure-stllib-0.3.tar.gz), cd to the pure-stllib-0.3 directory, and run `make`. After this you can (and should) also run `make check` to run a few unit tests to make sure that *pure-stlvec* and *pure-stlmap* work properly on your system. If `make check` works, run `sudo make install` to install *pure-stlvec* and *pure-stlmap*. Run `sudo make uninstall` to remove them.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix. Please see the Makefile for details.

9.3 Usage

pure-stlvec provides functions that act on a single mutable container, stlvec, which is a wrapper around C++'s vector, specialized to hold Pure expressions. It also provides functions that correspond to C++'s STL algorithms specialized to act on stlvecs.

pure-stlmap provides functions that act on six mutable containers, "stlmap", "stlset", "stlmmap", "stlmset", "stlhmap" and "stlhset", that are thin wrappers around the corresponding associative containers provided by C++, map, set, multimap, multiset, unordered_map and unordered_set, specialized to hold Pure expressions.

The functions provided by *pure-stlvec* and *pure-stlmap* are made available by importing one or more of the following modules.

`stlvec` - support for stlvecs

`stlvec::algorithms` - STL algorithms specialized to act on stlvecs

`stlmap` - support for stlmap and stlset

`stlmmap` - support for stlmmap and stlmset

`stlhmap` - support for stlhmap and stlhset

9.4 Documentation

Please see the documentation for *pure-stlvec* and *pure-stlmap*.

For the impatient, the functions that act on containers provided by the `stlmap`, `stlmmap`, `stlhmap` and `stlvec` modules are summarized in a rudimentary cheatsheet, pure-stllib-cheatsheet.pdf, which can be found in the pure-stllib/doc directory.

9.5 Changes

Version 0.1 - Bundle pure-stlvec-0.3 and pure-stlmap-0.1.

Version 0.2 - Bundle pure-stlvec-0.3 and pure-stlmap-0.2.

Version 0.3 - Bundle pure-stlvec-0.4 and pure-stlmap-0.3.

Chapter 10

pure-stlmap

Version 0.3, June 26, 2012

Peter Summerland <p.summerland@gmail.com>

pure-stlmap is a Pure interface to six associative containers provided by the C++ Standard Library: map, set, multimap, multiset, unordered_map and unordered_set.

10.1 Copying

Copyright (c) 2012 by Peter Summerland <p.summerland@gmail.com>.

All rights reserved.

pure-stlmap is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

pure-stlmap is distributed under a BSD-style license, see the COPYING file for details.

10.2 Introduction

This is pure-stlmap-0.1, the first release of pure-stlmap. It is possible that some of the functions might be changed slightly or even removed. Comments and questions would be especially appreciated at this early stage.

10.2.1 Supported Containers

The Standard C++ Containers Library, often refered to as the standard template library ("STL"), provides templates for generic containers and generic algorithms. pure-stlmap provides six mutable containers, "stlmap", "stlset", "stlmmap", "stlmset", "stlhmap" and "stlhset", that are thin wrappers around the corresponding associative containers provided by the STL, map, set, multimap, multiset, unordered_map and unordered_set, specialized to hold pure-expressions. pure-stlmap does not provide wrappers for unordered_multimap and unordered_multiset.

10.2.2 Interface

pure-stlmap provides a "key-based" interface that can be used to work with the supported STL containers in a way that should feel natural to Pure programmers. For example, the (!) function can be used to access values associated with keys and functions like `map`, `foldl`, `filter` and `do` can be used to operate on all or part of a container's elements without using an explict tail recursive loop. In addition, for the ordered containers, stlmap, stlmmap, stlset and stlmset, pure-stlmap provides an "interator-based" interface that corresponds to the C++ interface, mostly on a one-to-one basis.

The interface for the unordered or "hash table" containers, stlhmap and stlhset, is limited compared to that provided for the ordered containers. In particular iterators, operations on subsequences (ranges) and set operations are not supported.

In some cases, the STL's associative containers have different semantics than the the associative containers provided by the Pure standard library. Where there is a conflict, pure-stlmap follows the STL.

Many of the functions provided by pure-stlmap, such as the constructors, equivalence and lexicographical comparison operations, insert and erase operations, and the set operations are just thin wrappers around the the corresponding C++ functions. Users can consult the C++ Library documentation to understand the performance characteristics and corner case behavior of any pure-stlmap function that has a corresponding function in the STL.

The C++ library is sometimes more complicated than the Pure Standard Library. For example many of the applicable C++ functions, including set operations and tests for equality, assume that the containers are lexicographically ordered. The reward for playing by the rules (which occurs automatically for stlmap and stlset) is O(n) time complexity for comparison and set operations.

10.3 Installation

pure-stlmap-0.3 is included in the "umbrella" addon, *pure-stllib* which is available at http://code.google.com/p/pure-lang/downloads/list. After you have downloaded and installed *pure-stllib*, you will be able to use pure-stlmap (and *pure-stlvec*, as well).

10.4 Examples

The pure-stlmap/uts subdirectory contains Pure scripts that are used to test pure-stlmap. These scripts contain simple tests, each of which consists of a single line of code followed by a comment that contains the expected output. E.g.,

```
let sm1 = stlmap ["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5];
//- ()

sm1!stl::smbeg, sm1!"a", sm1!"d", sm1!"e"
//- 1,1,4,5

catch id $ sm1!"0";
//- out_of_bounds
```

You might consider pasting parts of these scripts into a temporary file that you can play with if you are curious about how something works.

Two short example programs, anagrams.pure and poly.pure, can be found in the pure-stlmap/examples subdirectory.

10.5 Quick Start

This section introduces the basic functions you need to get up and running with pure-stlmap. For a quick look at the other functions provided by pure-stlmap, you can refer to pure-stllib-cheatsheet.pdf, which can be found in the pure-stllib/doc directory.

10.5.1 Example Containers

The code snippets that appear in the examples that follow assume that six containers have been created by entering the following at the prompt.

```
$> pure -q
> using stlmap, stlhmap, stlmmap;
> using namespace stl;

> // Make some maps and sets with default characteristics
> let sm  = stlmap  ["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5];
> let shm = stlhmap ["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5];
> let smm = stlmmap ["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5];
> let ss  = stlset  ["a","b","c","d","e"];
> let shs = stlhset ["a","b","c","d","e"];
> let sms = stlmset ["a","b","c","c","d"];
```

The `using` statement imports the three modules provided by pure-stlmap: `stlmap` provides the interface for the stlmap and stlset containers, `stlmmap` provides the interface the stlmmap

and stlmset containers, and `stlhmap` provides the interface to the stlhmap and stlhset containers. The `let` statements set up an instance of each of the containers provided by pure-stlmap, loaded with some sample elements.

To save typing you can run readme-data.pure, a file that contains the corresponding source code. It can be found in in the pure-stlmap/examples directory.

10.5.2 Constructors

You can construct empty pure-stlmap containers using the `emptystlmap`, `emptystlset`, `emptystlmmap`, `emptystlmset`, `emptystlhmap` and `emptystlhset` functions.

```
> let sm1 = emptystlmap;   // uses (<) to order keys
```

You can construct a pure-stlmap container and fill it with elements all in one go using the `stlmap`, `stlset`, `stlmmap`, `stlmset`, `stlhmap` and `stlhset` functions.

```
> let shm1 = stlhmap ["a"=>1,"b"=>2,"c"=>3];

> members shm1;
["c"=>3,"a"=>1,"b"=>2]

> smh1!"b";
2
```

As opposed to the hashed containers (stlhmap and stlhset), the ordered containers (stlmap, stlset, stlmmap and stlmset) keep their elements ordered by key.

```
> let sm1 = stlmap ["a"=>1,"b"=>2,"c"=>3]; members sm1;
["a"=>1,"b"=>2,"c"=>3]
```

10.5.3 Ranges

For the ordered containers (stlmap, stlset, stlmmap and stlmset) you can work with subsequences, called "ranges", of the containers' elements. A range is specified by a tuple that consists of a container and two keys. If (sm, first_key, last_key) designates a range, the elements of the range are all of elements of the container sm whose keys are equivalent to or greater than first_key and less than last_key. If first_key and last_key are left out of the tuple, the range consists of all of sm's elements.

```
> members sm;               // no range keys - the whole container
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> members (sm,"b","e");     // a range from "b" up but not including "e"
["b"=>2,"c"=>3,"d"=>4]

> members (sm,"c1","z");    // keys do not have to be stored
["d"=>4,"e"=>5]
```

```
> members shm;              // works on a unordered set (with no range keys)
["c"=>3,"d"=>4,"e"=>5,"a"=>1,"b"=>2]
```

Two special keys, `stl::smbeg` and `stl::smend` are reserved for use in ranges to designate the first element in a container and the imaginary "past-end" element.

```
> members (sm,smbeg,"d");
["a"=>1,"b"=>2,"c"=>3]

> members (sm,"b",smend);
["b"=>2,"c"=>3,"d"=>4,"e"=>5]
```

Perhaps it should go without saying, but you cannot use either of these symbols as the keys of elements stored in a pure-stlmap container.

10.5.4 Inserting and Replacing Elements

You can insert elements and, for the maps (stlmap, stlmmap and stlhmap), replace the values associated with keys that are already stored in the map, using the `insert`, `replace` and `insert_or_replace` functions. For the maps, the elements to inserted are specified as (key=>value) hash-pairs.

```
> let sm1 = emptystlmap;

> insert sm1 ("e"=>5);              // returns number of elements inserted
1
> members sm1;
["e"=>5]

> replace sm1 "e" 15;               // returns value
15
> members sm1;
["e"=>15]

> catch id $ replace sm1 "x" 10;    // replace never inserts new elements
out_of_bounds

> insert sm1 ("e"=>25);             // insert never changes existing elements
0
> members sm1;
["e"=>15]

> insert_or_replace sm1 ("e"=>25);  // 1 value changed
1
> members sm1;
["e"=>25]
>
```

The `insert` and `insert_or_replace` functions are overloaded to insert or replace elements specified in a list, vector, stlvec or another pure-stlmap container (of the same type). E.g.,

```
> let sm2 = emptystlmap;

> insert sm2 ["b"=>2,"a"=>1];       // insert from a list
2

> insert sm2 (sm,"c","e");          // insert from a range
2

> members sm2;
["a"=>1,"b"=>2,"c"=>3,"d"=>4]

> insert_or_replace sm2 {"a"=>11,"e"=>15};
2

> members sm2;
["a"=>11,"b"=>2,"c"=>3,"d"=>4,"e"=>15]
```

10.5.5 Access

If you want to see if a key is stored in a container use the `member` function. (A key, k, is considered to be "stored" in a container if there is an element in the container that is equivalent to k.)

```
> member sm "x";  // ("x"=>val) is not an element of sm for any val
0

> member sm "a";  // ("a"=>1) is an element with key equivalent to "a"
1
```

The value (or values for a multi-key container) associated with a key can be accessed using the (!) function.

```
> sm!"a";   // return the value associated with "a"
1

> shm!"b";  // try it with a hashed map
2

> smm!"c";  // multimap returns a the list of values associated with "c"
[31,32]

> ss!"a";   // with sets, return the key
"a"

> sms!"c";  // with multisets, return a list of keys
["c","c"]
```

If the key is not stored in the container, (!) throws an `out_of_bounds` exception.

```
> catch id $ sm!"x"; // "x" is not stored as a key in sm
out_of_bounds
```

Please note that all access is strictly by keys. For example you cannot use the `member` function to determine if ("a"=>1) is an element stored in sm; you can only ask if the key "a" is stored in sm.

10.5.6 Erasing Elements

For any pure-stlmap container, you can use the `erase` function to remove all the elements associated with a given key in the container, all of the elements in the container or, unless the container is a stlhmap or stlhset, all of the elements in a range defined on the container.

```
> let shm1 = stlhmap shm;       // make some copies of maps
> let smm1 = stlmmap smm;
> let sm1 = stlmap sm;

> members smm1;                 // smm1 has multiple values for "c"
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> erase (shm1,"c");             // erase "c" keyed elements from a stlmmap
1
> members shm1;                 // all the "c" keyed elements are gone
["d"=>4,"e"=>5,"a"=>1,"b"=>2]

> erase shm1;                   // erase all elements
4
> empty shm1;
1

> erase (sm1,"b","d");         // erase a subsequence
2
> members sm1;
["a"=>1,"d"=>4,"e"=>5]

> erase (sm1,"x");             // attempt to erase something not there
0

> erase (smm1,"c");            // erase all elements with key "c"
2
> members smm1;
["a"=>1,"b"=>2,"d"=>4,"e"=>5]
```

10.5.7 Conversions

The elements of an associated container be copied into a list, vector or stlvec using the `members`, `stl::vector` and `stlvec` functions. For ordered containers (stlmap, stlset, stlmmap and stlmset) the list, vector or stlvec can be built from a range.

```
> members ss;
["a","b","c","d","e"]

> members (ss,"b","d"); // list subsequence from "b" up to but not "d"
["b","c"]

> members (smm,"c","e");
["c"=>31,"c"=>32,"d"=>4]

> members (shm,"b","d"); // fails - ranges not supported for stlhmaps
stl::members (#<pointer 0x83b4908>,"b","d")

> members shm;          // ok - all elements are copied
["d"=>4,"e"=>5,"a"=>1,"b"=>2,"c"=>3]

> vector (sm,smbeg,"d");
{"a"=>1,"b"=>2,"c"=>3}

> using stlvec;
> members $ stlvec sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]
```

You can convert the contents of an ordered container (stlmap, stlset, stlmmap or stlmset) or a range defined on one to a stream using the `stream` function.

```
> let ss1 = stlhset (0..100000);

> stats -m

> let xx = drop 99998 $ scanl (+) 0 (stream ss);
0.3s, 18 cells

> list xx;
[704782707,704882705,704982704,705082704]
0s, 17 cells
```

10.5.8 Functional Programming

Most of the Pure list operations, including `map`, `do`, `filter`, `catmap`, `foldl` and `foldl1` can be applied to any of pure-stlmap's associative containers. E.g.,

```
> map (\x->x-32) shs;
["D","E","A","B","C"]

> using system;

> do (puts . str) (sm,smbeg,"c");
"a"=>1
"b"=>2
()
```

List comprehensions also work.

```
> [k-32=>v+100 | (k=>v) = smm; k>"a" && k<"e"];
["B"=>102,"C"=>131,"C"=>132,"D"=>104]

> {k-32=>v+100 | (k=>v) = (smm,"b","e")};
{"B"=>102,"C"=>131,"C"=>132,"D"=>104}
```

It is highly recommended that you use the functional programming operations, as opposed to recursive loops, whenever possible.

10.6 Concepts

This section describes pure-stlmap's containers, iterators, ranges, elements, keys, values and how these objects are related to each other. It also describes a group of functions associated with containers that help define the container's behavior. E.g., each ordered container (stlmap, stlset, stlmmap or stlmset) stores a function that it used to order its keys and to determine if two keys are equivalent.

10.6.1 Containers and Elements

The six associative containers supported by pure-stlmap can be grouped together in terms of certain defining attributes.

The three "maps" provided by pure-stlmap, stlmap, stlmmap and stlhmap, associate values with keys. If a value v is associated with a key, k, in an map, m, then we say that (k=>v) is an element of m, k is a key stored in m and v is a value stored in m.

The three "sets" provided by pure-stlmap, stlset, stlmset and stlhset, hold single elements, as opposed to key value pairs. If an element e is contained a set, s, we say that e is simultaneously an element, key and value stored s. In other words, we sometimes speak of a set as if it were a map where each element, key and value are the same object.

The "ordered" containers, stlmap, stlset, stlmmap and stlmset, each have a "key-less-than" function that they use keep their elements in a sequence that is ordered by keys. The default key-less-than function is `(<)`, but this can be changed when the container is created. The elements stored in a stlmap or stlset have unique keys, i.e., two elements stored in the container will never have equivalent keys. For these purposes, two keys are "equivalent" if neither key is key-less-than the other. In contrast, stlmmap and stlmset do not have unique keys. I.e., it is possible for different elements stored in a stlmmap or stlmset can have equivalent keys.

The "hashed" containers, sthmap and stlhset do not keep their elements in a sequence. Instead they store their elments in a hash table using a "key-hash" function and a "key-equal" function. Currently the key-hash function is always `hash` and the key-equal function is always (===), both of which are defined in the Prelude. The elements stored in a hashed container have unique keys. I.e., two elements stored in the container will never by "key-equal". At times we say that two keys stored in a hashed container are "equivalent" if they are key-equal.

The "ordered maps", stlmap and stlmmap, each have a "value-less-than" function and a "value-equal" function that is used for lexicographical comparisons. The default functions are (<) and (==) respectively, but these can customized when the container is created.

As is the case for the underlying C++ functions, set operations (i.e., union, intersection, etc.) and container equivalence for the ordered containers are based on lexicographical comparisons. For these purposes one element, e1, is less than another, e2, if (a) e1's key is less-than e2's key and, (b) if the ordered container is a stlmap or stlmap, e1's value is value-less-than e2's value. Finally, for purposes of determining if two ordered containers are equal, e1 and e2 are considered to be equal if (a) their keys are equivalent and (b), in the case of stlmap or stlmmap, their values are value-equal.

Set operations are not provided for the hashed containers, stlhmap and stlhset.

10.6.2 Ranges

For the ordered containers (stlmap, stlset, stlmmap and stlmset), you can work with a subsequence or "range" of a container's elements. Given an ordered container, oc, and keys f and l, the range (oc,f,l) consists of all of the elements in oc starting with the first element that is not less than f up to but not including the first element that is greater or equal to l. Note that f and l do not have to be stored in oc.

```
> members (sm,"b","e");
["b"=>2,"c"=>3,"d"=>4]

> members (sm,"c1",smend);
["d"=>4,"e"=>5]
```

When a range is passed to a function provided by pure-stlmap, the keys can be dropped, in which case the range consists of all of the container's elements.

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]
```

Please note that support for ranges is not provided for the unordered containers (stlhmap and stlhset). Most pure-stlmap functions that act on ranges can, however, operate on stlhmaps or stlhsets as well, except that, for stlhmaps and stlhsets, they always operate on all of the container's elements. Accordingly, whenever the documentation of a function refers to a range, and the container in question is a a stlhmap or stlhset, the range simply refers to the container itself.

10.6.3 Iterators

The native STL interface is based on "iterators" that point to elements in containers. pure-stlmap provides support for iterators defined on its ordered containers (stlmap, stlmmap, stlset and stlmset) but not for its unordered containers (stlhmap and stlhset).

Iterators are most useful when dealing with stlmmaps where elements with different values can have equivalent keys. In most cases, it is recommended that you avoid using iterators.

The functions that operate on or return iterators are discussed separately at the end of this document.

10.6.4 Selecting Elements Using Keys

Throughout pure-stlmap, unless you resort to using iterators, you can only specify elements and ranges of elements using keys. For example you cannot use the `member` function to see if a specific key, value pair is an element of a stlmap.

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> member sm "a";
1

> catch id $ member sm (a=>1);
bad_argument
```

In the last line of code, `member` treats (a=>1) as a key. Because (a=>1) cannot be compared to a string using `(<)`, the ersatz key is treated as a bad argument.

This "key access only" approach can be an issue for stlmmaps and because multiple elements can have equivalent keys. I.e., given a stlmmap, smm, that containes multiple element with keys equivalent to, say, k, which element should (!) return? pure-stlmap dodges this issue by returning all on them. Thus, for stlmmap and stlmset (!) and `replace` work with lists of elements associated with a given key rather than, say, the first elment with the given key.

```
> members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4];

> smm!"c";
"c"=>[31,32]

> replace smm "c" [31,32,33]; members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"c"=>33,"d"=>4]

> replace smm "c" []; members smm;
["a"=>1,"b"=>2,"d"=>4,"e"=>5]
```

If selecting and replacing lists of elements with the same key is not convenient, you can always use iterators to track down and modify any specific element.

10.6.5 C++ Implementation

For those that want to refer to the C++ standard library documentation, stlmap is (essentially) map<px*,px*>, stlmmap is multimap<px*,px*> and stlhmap is unordered_map<px*,px*>, where px is defined by "typedef pure_expr px". I.e., in C++ Containers library speak, key_type is px*, mapped_type is px* and value_type is pair<px*,px*>. This might be a bit confusing because pure-stlmap's (key=>value) "elements" correspond

to C++ value_types, a pair<key_type,mapped_type>, and pure-stlmap's values correspond to mapped_types. The C++ objects for stlset, stlmset and stlhset are the same as stlmap, stmmap and stlhmap except that pure-stlmap ensures that the second member of the C++ value_type pair is always NULL.

10.7 Modules

pure-stlmap provides three separate modules `stlmap`, `stlmmap` and `stlhmap`.

Importing any one of these modules defines the stl namespace as well as two important symbols, `stl::smbeg` and `stl::smend`.

constructor `stl::`**smbeg**
constructor `stl::`**smend**
: These symbols are used to designate the key of the first element in an ordered container (stlmap, stlset, stlmmap or stlmset) and the key of an imaginary element that would come immediately after the last element of in the constainer. They are used to define ranges over the ordered containers.

E.g.,

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> members (sm,"c",smend);
["c"=>3,"d"=>4,"e"=>5]
```

10.7.1 The stlhmap Module

If all you want is fast insertion and lookup, you don't care about the order of the elements stored in the container, and you do not want to use set operations like `stl::map_intersection`, then `stlhmap` is probably your best choice. The supported containers, stlhmap and stlhset are simpler to use and faster than the other containers provided by pure-stlmap.

The `stlhmap` module defines stlhmaps and stlhsets and provides functions for dealing with them. You can import it by adding the following `using` statement to your code.

```
> using stlhmap;
```

The `stlhmap` module defines types two types:

type `stlhmap`
type `stlhset`

Please note that a stlhset is just a stlhmap where the values associated with keys cannot be accessed or modified. I.e., a stlhset is a specialized kind of stlhmap.

10.7.2 The stlmap Module

The `stlmap` module provides you with stlmaps and stlsets and the functions that operate on them. Consider using these containers if you want their elements to be orderd by key, want to use ranges or if you are using any set operations (`stl::map_union`, `stl::map_intersection`, etc).

You can import the stlmap module by adding the following using statement to your code.

```
> using stlmap;
```

Importing the stlmap module introduces types to describe stlmap and stlset, their iterators and ranges defined on them.

type stlmap
type stlset

type stlmap_iter

type stlmap_rng

Please note that a stlset is just a stlmap where the values associated with keys cannot be accessed or modified. I.e., a stlset is a specialized kind of stlmap. Accordingly, it is not necessary, for example, to define a separate type for iterators on stlsets as opposed to iterators on stlmaps.

10.7.3 The stlmmap Module

If you need a multi-keyed container, the `stlmmap` module, which provides support for stlmaps and stlmsets, is your only choice. Set operations and ranges are supported, but the semantics are more complicated than is the case for stlmap and stlset. Because the keys stored in multi-keyed containers are not unique you might have to resort to using iterators when working with them.

You can import the `stlmmap` module by adding the following using statement to your code.

```
> using stlmmap;
```

Importing the stlmmap module introduces types to describe stlmmap and stlmset, along with their iterators and ranges defined on them.

type stlmmap
type stlmset

type stlmmap_iter

type stlmmap_rng

Please note that a stlmset is just a stlmmap where the values associated with keys cannot be accessed or modified. I.e., a stlmset is a specialized kind of stlmmap. Accordingly, it is not necessary, for example, to define a separate type for iterators on stlmsets as opposed to iterators on stlmmaps.

10.8 Container Operations

Each of the six associative containers supported by pure-stlmap has its own set of unique characteristics. Because of this the description of functions that operate on more than one type of container can get a little complicated. When reading this section it might be helpful to consult pure-stllib-cheatsheet.pdf which can be found in the pure-stlib/doc directory.

10.8.1 Container Construction

New empty ordered containers (stlmap, stlset, stlmmap and stlmset) can be constructed using optional parameters that allow you to specify customized key-less-than functions, default values, value-less-than and value-equal functions.

mkstlmap (klt,dflt,vlt,veq)
mkstlmmap (klt,dflt,vlt,veq)
: Create a new stlmap or stlmmap where `klt` is the map's key-less-than function. dflt is the maps default value (used by replace_with and find_with_default). vlt is the map's value-compare function and veq is its value-equal function. Only `klt` is required, and the default values for dflt, vlt, veq are [], (<) and (==) respectively.

mkstlset klt
mkstlmset klt
: Create a new stlset or stlmset where `klt` is the set's key-less-than function.

The internal lookup functions for the ordered containers (stlmap, stlset, stlmmap and stlmset) are optimized to avoid callbacks if the container's key-less-than function is is `(>)` or `(<)` and the keys being compared are a pair of strings, ints, bigints or doubles.

You can create an empty associative container using default values for using `emptystlmap` and friends.

emptystlmap
emptystlmmap
emptystlset
emptystlmset
: Create a new ordered map or set using default values. I.e., emptystlmap is the same as mkstlmap `(<)`, and so on.

emptystlhmap
emptystlhset
: Create a new stlhmap or stlhset with default values. The hash-function is hash and the value-equal function is (===).

Convenience functions are also provided to construct an empty container and insert elements into it in one go. The source of the elements can be a list, vector, a stlvec, or a range defined on another container of the same type as the new container.

stlmap src
stlmmap src

stlset src
stlmset src
stlhmap src
stlhset src
: Create an associative constructor using default values and insert elements from copied from `src`. `src` can be a list, vector or stlvec of elements or a range defined over a container of the same type as the new container. If the new container is a stlmap, stlmmap or stlhmap, the elements of src must be (key=>val) pairs. If the new container is a stlset, stlmset or stlhset they can be any pure expression that can be used as a key (i.e., anything except for `stl::smbeg` or `stl::smend`).

10.8.2 Information

This group of functions allows you make inquiries regarding the number of elments in a container, the number of instances of a given key held by a container, the upper and lower bounds of a range and other information. In addition this group includes a function that can be used to change the number of slots used by a stlhmap or stlhset.

acon
: Return the number of elements in `acon`.

`stl::`**empty** acon
: Return true if `acon` is empty, else false.

`stl::`**distance** rng
: Returns the number of elements contained in `rng` where rng is a range defined on an ordered container (stlmap, stlmmap, stlset, stlmset).

`stl::`**count** acon k
: Returns the number of elements in an associative container, acon, that have a key that is equivalent to `k`.

`stl::`**bounds** rng
: Return a pair of keys, first and last, such that first <= k < last for each k, where k is the key of an element in `rng`. If there is no such last, the second member of the returned pair will be `stl::smend`. If first is the key of the first element of `rng's` container, the first member of the returned pair will `stl::smbeg`.

Here are two examples using the `stl::bounds` function. Notice that bounds returns `stl::smbeg` instead of "a" in the first example.

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> bounds sm;
stl::smbeg,stl::smend

> bounds (sm,"a1","e");
"b","e"
```

`stl::`**`container_info`** acon
: If `acon` is a stlmap or stlmmap, returns (0, klt, dflt, vlt, veq) where klt is `acon`'s key-less-than function, dflt is its default value, vlt is its value-less-than function and veq is its value_equal function. If `acon` is a stlset or stlmset, returns (1,klt,_,_,_) where klt is `acon`'s key-less-than function. If `acon` is a stlhmap or stlhset, returns (is_set, bucket_count, load_factor, max_load_factor).

`stl::`**`bucket_size`** hacon n
: Returns the number of elements in `hacon`'s nth (zero-based) bucket where `hacon` is a stlhmap or stlhset.

`stl::`**`hmap_reserve`** hacon mlf size
: Sets `hacon`'s max_load_factor to `mlf`, sets the number of `hacon 's buckets to ''size/mlf' and rehashes ''hacon` where `hacon` is a stlhmap or stlhset.

10.8.3 Modification

You can insert new items or, for the maps (stlmap, stlmmap and stlhmap), replace values associated with keys using the `insert`, `replace` or `insert_or_replace` functions.

Please note that when working with the ordered containers (stlmap, stlset, stlmmap and stlmset) the keys of elements passed to these functions must be compatible with the container's key-less-than function and keys that are already inserted. E.g.,

```
> members ss;
["a","b","c","d","e"]

> catch id $ insert ss 1;   // e.g., 1<"a" is not defined
bad_argument
```

Currently there is no similar restriction for stlhmaps and stlhsets because (a) they do not have a key-less-than function and (b) the function they do use for testing equality, the key-equal function is always (===), a function that can compare any two objects.

```
> members shs;
["c","d","e","a","b"]

> insert shs 1;
1
> members shs;
["c",1,"d","e","a","b"]
```

Elements can be inserted into a pure-stlmap container individually or en masse from a list, vector, stlvec or another container of the same type. If there is a key in the container that is equivalent to the key of the element being inserted, the element will not be inserted (unless the container is a stlmmap or stlmset, both of which can hold multiple elements with equivalent keys).

`insert` acon src
: Attempts to copy elements from `src` a valid "insert source" into `acon` which can be any

pure-stlmap container. A valid insert source is (a) a single element, (b) a list, vector, stlvec of elements or (c), a range over an associative container of the same type as `acon`. If `acon` is an associative map (stlmap, stlmmap or stlhmap), the `src` itself, or all the elements of `src`, must be key value pairs of the form (k=>v). In contrast, if `acon` is a stlset, stlmset or stlhset, `src` or all of its elements can be any pure object (except `stl::smbeg` or `stl::smend`). If `acon` is a stlmap, stlset, stlhmap or stlhset, the element will not be inserted if its key is already stored in the target container. Returns the number of elements inserted, if any.

If you are dealing with a stlmap or stlhmap and want to override the values of elements have keys that equivalent to the keys of the items you wan to insert you can use the `insert_or_replace` function.

`insert_or_replace` acon src
: The same as `insert` except that (a) `acon` must be a stlmap or a stlhmap and (b) if an element (key=>newval) is about to be inserted and the container already contains an element (key=>oldval) the element in the container will be changed to (key=>newval). Returns the number of elements inserted or updated.

`replace` map key x
: `map` must be a stlmap, stlmmap or stlhmap. If `key` is not stored in `map` this function throws `out_of_bounds`. If `map` is a stlmap or stlhmap and (oldkey=>oldval) is an element of `map`, where oldkey is equivalent to `key`, change the element to (oldkey=>''x''). If `map` is a stlmmap and `key` is stored in `map`, change the values of elements with key eqivalent to `key`, one by one, to the elements of `x`. Add or delete elements as necessary so that, when the smoke clears, the values of `map`!''key'' are copies of the elements of `x`. In all cases, if `key` is stored in `map` returns `x`.

Here are some examples using `replace`.

```
> members sm1;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> replace sm1 "e" 50;
50

> members sm1;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>50]

> members smm1;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> replace smm1 "c" [31,33,35,36] $$ smm1!"c";
[31,33,35,36]

> replace smm1 "c" [] $$ smm1!"c";
[]

> members smm1;
["a"=>1,"b"=>2,"d"=>4,"e"=>5]
```

replace_with fun map (k=>v)
: map must be a stlmap. The effect of this function is as follows: (a) if ~ member map k then insert map (k``=>dflt) else (), where dflt is ``map's dflt value, (b) replace map k nv when nv = fun v (map!``k``) end. Returns map.

Here is an example using replace_with in which a stlmmap is converted to a stlmap.

```
> let sm1 = emptystlmap;

> members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> do (replace_with (:) sm1) smm;
()

> members sm1;
["a"=>[1],"b"=>[2],"c"=>[32,31],"d"=>[4],"e"=>[5]]
```

Here is another example in which items are counted.

```
> let sm1 = mkstlmap ( (<), 0 );

> members sms;
["a","b","c","c","d"]

> do (\x->replace_with (+) sm1 (x=>1)) sms;
()

> members sm1;
["a"=>1,"b"=>1,"c"=>2,"d"=>1]
```

You can remove all the elements in a container, remove all the elements equivalent to a given key or a remove a range of elements using the erase function.

erase acon
erase (acon,k)
erase (acon,k1,k2)
: The first form erases all elements in acon which can be any container provided by pure-stlmap. The second erases all elements in acon with key equivalent to k. The third erases the elements in the range (acon,``k1'',``k2''). The third form only applys to the ordered containers (stlmap, stlmmap, stlset and stlmset), not stlhmap or stlhset (because ranges are not defined for stlhmaps or stlhsets). Returns the number of elements removed from the container.

Here are some examples using erase.

```
> members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> erase (sm,"z");
0

> erase (smm,"c");
```

```
2

> members smm;
["a"=>1,"b"=>2,"d"=>4,"e"=>5]

> erase (smm,"b","e");
2

> members smm;;
["a"=>1,"e"=>5]
```

`stl::`**`swap`** acon1 acon2
: Swaps the elements of the two containers, `acon1` and `acon2` where `acon1` and `acon2` are the same type of container (E.g., both are stlmaps or both are stlmsets).

10.8.4 Accessing Elements

You can test if a key is stored in a container and access the value associated with a key using the familiar `member` and (!) functions.

`member` acon k
: Returns true if `acon`, any container provided by pure-stlmap, contains an element that has a key that is equivalent to `k`.

acon **`!`** k
: If `acon` is not a stlmmap then (a) if `acon` has an element with key equivalent to `k` return its value, otherwise (b) throw an `out_of_bounds` exception. If `acon` is a stlmmap then (a) if acon has as least one element with key equivalent to `k` return a list of values of all the elements with key equivalent to `k`, otherwise (b) return an null list.

E.g.:

```
> sm!"c";
3

> catch id $ sm!"f";    // "f" is not stored in sm
out_of_bounds

> catch id $ sm!100;    // 100 cannot be compared to strings using (<)
bad_argument

> smm!"c";              // for stlmmap, return list of values
[31,32]

> smm!"f";              // stlmmap returns null list if key is not stored
[]
```

You can access a sequence of elements in an ordered container (stlmap, stlset, stlmmap or stlmset) without resorting to iterators using the next_key and prev_key functions.

`stl::`**`next_key`** acon k

`stl::`**`prev_key`** acon k

: `acon` must be a stlmap, stlset, stlmmap or stlmmap. Also if `k` is not `stl::smbeg`, `stl::smend` or an element of acon an `out_of_bounds` exception will be throw. `next_key` returns the key of the first element in acon that has a key that is greater than `k`. If no such element exists or if `k` is `stl::smend`, returns `stl::smend`. `prev_key` returns the last element in acon that has a key that is less that `k`, or, if no such element exists, throws an `out_of_bounds` exception.

For various reasons, it is very common to see a call to (!) or `replace` preceded by a call to `member` with the same container and key. E.g.,

```
> bump_wc sm w = if member sm w then replace sm w (sm!w + 1)
                 else insert sm (w=>1);
```

In general, this function would require two lookups to add a new word and three lookups to bump the count for an existing word. For the ordered containers, lookups have O(log N) complexity which can be relatively slow for large containers.

To speed things up, each stlmap or stlset maintains a small cache of (key, C++ iterator) pairs for recently accessed keys. During lookup, the cache is checked for a matching key, and if the key is found, the element pointed to by the C++ iterator is used immediately. Thus, when applied to a stlmap or stlset bump_wc will use only one O(log N) search, rather than two or three. For these purposes, a key matches a key in the cache only if it is the same Pure object (i.e., the test is C++ pointer equality, not Pure's (===) or (==) functions). For example, the following will result in two O(log N) lookups.

```
> if member sm "a" then sm!"a" else insert sm ("a"=>10);
```

Here each "a" is a distinct Pure object. The two "a"s satisfy (==) and even (===) but they are not the same internally and the caching mechanism will not help.

Almost any pure-stlmap function that accepts a stlmap or stlset as an argument will check the container's cache before doing an O(log N) lookup. Currently the cache is limited to hold only the most recently used key.

Here are some examples produced by compiling pure-stlmap with a trace function that shows caching in action.

```
> let a_key = "a";

> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> member sm a_key;           // a_key is not yet in the cache
1

> sm!a_key;                  // a_key is found in the cache
found iterator for: "a"
1

> replace sm a_key 10;
found iterator for: "a"
```

```
10

> sm!"a";                    // "a" is a new key, not same C++ pointer as k or a_key
1

> let k = next_key sm a_key; // now k is in the cache, in front of a_key
found iterator for: "a"

> let k1 = next_key sm k;    // now k1 is at the head of the queue
found iterator for: "b"

> replace sm k1 30;
found iterator for: "c"
30

> members sm;
["a"=>10,"b"=>2,"c"=>30,"d"=>4,"e"=>5]
```

These examples show that caching can be effective wnen visiting elements of a stlmap or stlset in order using `next_key` or `prev_key`.

10.8.5 Conversions

The contents of a pure-stlmap container can be copied to a list, vector, stlvec. For stlmaps, stlsets, stlmmaps and stlmsets, these operations act on ranges as well as on the entire container.

members rng
: Returns a list of the elments in the range, `rng`.

keys rng
vals rng
: Return the keys and vals of the range's elements.

Here are some examples using the `members`, `keys` and `vals` functions.

```
 > members shm;   // must do all of shm elements because shm is a stlhmap
["d"=>4,"e"=>5,"a"=>1,"b"=>2,"c"=>3]

> keys (sm,"b","e"); // can ask for a range - sm is an ordered container
["b","c","d"]

> vals (sm,"b","e");
[2,3,4]
```

`stl::`**vector** rng
: Return a vector containing the elments of in the range, rng.

stlvec rng
: returns a stlvec containing the elments of in the range, rng.

You can also convert an ordered container (stlmap, stlset, stlmmap or stlmset) into a stream of elements.

stream rng
: Returns a stream consisting of the range's elements.

Here is an example using the stream function on a stlmmap.

```
> members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> take 3 $ stream smm;
("a"=>1):#<thunk 0xb70f438c>

> list ans;
["a"=>1,"b"=>2,"c"=>31]
```

10.8.6 Functional Programming

pure-stlmap provides the most commonly used functional programming operations, implemented to act on ranges as if they were lists.

do fun rng
map fun rng
filter pred rng
foldl fun x rng
foldl1 fun rng
foldr fun x rng
foldr1 fun rng
: These functions are the same as the corresponding functions provided in the Prelude for lists. `rng` is a rng defined on a stlmap, stlset, stlmmap or stlmset or `rng` is simply a stlhmap or stlhset. `foldr` and `foldr1` are not defined for stlhmaps or stlhsets.

Here are some examples.

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> map (\(k=>v)->k+str v) (sm,"b","e");
["b2","c3","d4"]

> foldr1 (\(k=>v) (ks=>sum)-> (k+ks=>v+sum)) (sm,"b","e");
"bcd"=>9

> filter (\(k=>v)->v mod 2) sm;
["a"=>1,"c"=>3,"e"=>5]
```

listmap fun rng
catmap fun rng
rowmap fun rng
rowcatmap fun rng

colmap fun rng
colcatmap fun rng
: These functions are the same as the corresponding functions provided in the Prelude for lists. `rng` is a rng defined on a stlmap, stlset, stlmmap or stlmset or simply a stlhmap or stlhset.

These functions are provided primarily to enable the use of list and matrix comprehensions over pure-stlmap's containers. E.g.,

```
> [ k + str v | (k=>v) = (sm,"b","e")];
["b2","c3","d4"]

> [ k=>v | (k=>v) = sm; v mod 2];
["a"=>1,"c"=>3,"e"=>5]

> { {k;v} |  (k=>v) = sm; v mod 2};
{"a","c","e";1,3,5}
```

The functional programming operations work directly on the underlying data structure.

```
> let ints = 0..10000;

stats -m
> filter (==99) ints;
[99]
0s, 6 cells
```

10.8.7 Comparison

Two associative containers of the same type are considered to be equal if they contain the same number of elements and if each pair of their corresponding elements are equal. Two elements are equal if their keys are equivalent and, if the container is a stlmap, stlmap or stlhmap, the values associated with equal keys are equal (using the container's value-equal function).

`stl::`**`map_equal`** rng1 rng2

rng1 **==** rng2
rng1 **~=** rng2
: Test `rng1` and `rng2` for equality or nonequality where `rng1` and `rng2` are ranges defined over containers of the same type.

You need to be careful when using these operators. E.g.,

```
> members ss;
["a","b","c","d","e"]

> let xx = stlset ss;

> xx == ss;
1
```

```
> (xx,"a","c") == (ss,"a","c");  // oops!
0
```

The second comparison was intended to compare identical ranges and return true. It failed to do so because (==) is defined in the Prelude to compare tuples element by element, long before it is defined in the stlmap module to compare ranges. The tuple operation take precedence and determines that the tuples are not equal because `xx` and `ss` are different (pointers) for purposes of this comparison. To avoid this issue when using ranges, you can use the `stl::map_equal` function.

> map_equal (xx,"a","c") (ss,"a","c"); 1

The other comparison operators `(<)`, `(<=)`, `(>)` and `(>=)` are provided only for the ordered containers (stlmap, stlset, stlmmap and stlmset). These operators reflect lexicographical comparisons of keys and, then if the keys are equal, lexicographical comparisons of values. I.e., this is not set inclusion - order matters. Accordingly, these comparison operators are not defined for a stlhmap or stlhset.

rng1 **<** rng2
: Traverse the ranges comparing pairs of elements e1 and e2. If e1 is less than e2, stop and return true; if e2 is less than e1 then stop and return false. If rng1 is exhausted but rng2 is not, return true, else return false. The two ranges must be defined on ordered associative containers of the same type.

rng1 **>** rng2
rng1 **<=** rng2
rng1 **>=** rng2
: The these three operators are the same as `rng2 < rng1`, ~(`rng1`>‘‘rng2‘) and ~(`rng1‘‘<‘‘rng2`) respectively.

You also have to be careful when using equivalence and comparison operators with stlmmaps because elements with the same key and different values are not necessarily ordered by values.

```
> let smm2 = stlmmap ["a"=>1,"b"=>2,"c"=>32,"c"=>31,"d"=>4];

> members smm;
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4]

> members smm2;
["a"=>1,"b"=>2,"c"=>32,"c"=>31,"d"=>4]

> smm == smm2; // probably not what you want
0
```

These operations do not make much sense for a stlmmap unless elements with equivalent keys are stored by value, in the order enforced by the stlmmap's value-comp function. In this regard it is worth noting that, depending on your implementation, the `insert` function may or may not preserve the order of insertion of elements with equivalent keys (C++11 does preserve the order).

10.8.8 Set Algorithms

pure-stlmap provides wrappers for the STL set algorithms that apply to ranges defined on the four ordered associative containers (stlmap, stlset, stlmmap and stlmset). These algorithms are very efficient, with linear time complexity, but they do require that the elements of the two ranges be ordered. Accordingly, the set algorithms are not applicable to stlhmap or stlhset. Also, when dealing with stlmmaps, care must be taken to ensure that items with the equivalent keys are ordered by their values.

`stl::`**`map_merge`** rng1 rng2
: Constructs a new ordered container from `rng1` and then insert the elments of `rng2` into the new container and return it. `rng1` and `rng2` must be defined on the same type of ordered container.

`stl::`**`map_union`** rng1 rng2
`stl::`**`map_difference`** rng1 rng2
`stl::`**`map_intersection`** rng1 rng2
`stl::`**`map_symmetric_difference`** rng1 rng2
`stl::`**`map_includes`** rng1 rng2
: Returns a new ordered associative container of the same type as the ordered containers underlying `rng1` and `rng2`. If the ranges are defined over a stlmap or stlmmap elements of `rng1` have priority over the elments of `rng2`. Uses `rng1`'s key-less-than, value-less-than and value-equal functions.

pure-stlmap's set functions do not necessarily produce the same results as their Pure standard library counterparts. In particular, when applied to multi-keyed contaners, `stl::map_union` Produces the multiset union of its arguments while (+) in the Pure standard library produces the multiset sum. If you want the multiset sum of a stlmmap or stlhmap, use `stl::map_merge`. Also, in pure-stlmap, as in the STL, the left hand map or set has priority of elements while in the Pure standard library the right hand set has priority of elements. This can make a difference when applying set operations to a pair of stlmaps or stlmmaps. E.g.,

```
> let smm1 = stlmmap ["a"=>1,"b"=>2,"c"=>31,"c"=>32];
> let smm2 = stlmmap ["c"=>32,"c"=>32,"c"=>33,"d"=>4,"e"=>5];

> members $ map_merge smm1 smm2; // three "c"=>32
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"c"=>32,"c"=>32,"c"=>33,"d"=>4,"e"=>5]

> members $ map_union smm1 smm2;  // two "c"=>32
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"c"=>32,"c"=>33,"d"=>4,"e"=>5]

> let sm1 = stlmap ["a"=>1,"b"=>2,"c"=>31];
> let sm2 = stlmap ["c"=>32,"d"=>4,"e"=>5];

> members $ map_union sm1 sm2; // "c"=>31 from sm1, not "c"=>32 from sm2
["a"=>1,"b"=>2,"c"=>31,"d"=>4,"e"=>5]

> members $ map_intersection sm1 sm2; // "c"=>31 from sm1
["c"=>31]
```

10.8.9 Direct C Calls

It is common to encounter code that (a) tests if a key is stored in a container using `member` and (b) in the case of maps, retreives the value or values associated with the key using (!) and/or (c) changes the value or values using `replace`. Depending on what modules have been loaded, these functions may be heavily overloaded which can cause a small delay when the functions are called. To avoid this, pure-stlmap exposes the corresponding C functions so that they can be called directly. The C functions have the same name as the overloaded functions except for a prefix. E.g.,

`stl::`**`sm_member`** sm key
`stl::`**`sm_get`** sm key
`stl::`**`sm_put`** sm key val
: The first two functions are the direct C call equivalents of (`::member sm key`) and (`sm!key`). The third is like (`::replace sm key val`) except that it will insert (key=>val) if key is not already stored in `sm`. Here, `sm` is a stlmap or a stlset (except that sm_put is not defined for stlsets).

`stl::`**`shm_member`** shm key
`stl::`**`shm_get`** shm key
`stl::`**`shm_put`** shm key val
: The first two functions are the direct C call equivalents of (`::member shm key`) and (`shm!key`). The third is like (`::replace shm key val`) except that it will insert (key=>val) if key is not already stored in `shm`. Here, `shm` is a stlhmap or a stlhset (except that shm_put is not defined for stlhsets).

`stl::`**`smm_member`** smm key
`stl::`**`smm_get`** smm key
`stl::`**`smm_put`** smm key vals
: The first two functions are the direct C call equivalents of (`::member smm key`) and (`smm!key`). The third is like (`::replace smm key val`) except that it will insert (key=>val1, key=>val2, ...) if key is not already stored in `smm`. Here, `smm` is a stlmmap or a stlmset (except that smm_put is not defined for stlmsets).

10.9 Iterators

This section provides a quick overview of pure-stlmap's "iterator-based" interface.

10.9.1 Concepts

Given a valid iterator you can access, modify or erase the element it points to.

```
> let sm1 = stlmap sm; members sm1;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5];

> let i = find sm1 "b";   // use find to get an iterator - like C++
```

```
> get_elm i;
"b"=>2

> get_val i;
2

> put_val i 20;
20

> members sm1;
["a"=>1,"b"=>20,"c"=>3,"d"=>4,"e"=>5]
```

Please note that you can never modify an element's key, only its value. If you want to change both key and value, you have to erase the element and insert a new element.

```
> erase (sm1,i) $$ insert sm1 ("b1"=>21);
1

> members sm1;
["a"=>1,"b1"=>21,"c"=>3,"d"=>4,"e"=>5]
```

Given two iterators, i and j, pointing into a ordered container oc, the range (i,j), denotes oc's elements starting with "oc[i]", the element pointed to by i, up to but not including oc[j]. In pure-stlmap, this range is denoted by the tuple (i,j).

```
> members sm;
["a"=>1,"b"=>2,"c"=>3,"d"=>4,"e"=>5]

> let i = stl::find sm1 "b"; // get the iterator

> let j = stl::find sm1 "e";

> members (i,j);             // get the elements in the range
["b"=>2,"c"=>3,"d"=>4]
```

Perhaps it is worth mentioning that functions that act on ranges do not care if the range is specified by a pair of iterators or by keys.

```
> members ss;
["a","b","c","d","e"]

> map (+21) (ss,"c",smend);
["x","y","z"]

> let i = find ss "c";
> let j = pastend ss;
> map (+21) (i,j);
["x","y","z"]
```

10.9.2 Exceptions

In pure-stlmap functions that accept iterators throw a `bad_argument` exception if called with an invalid iterator. An iterator remains valid until the element it was pointing to has been erased. These functions also attempt to throw bad argument exceptions for invalid usage that would otherwise result in undefined behavior. An example of an invalid use would be a range specified by iterators from different containers. Here are some examples of iterator errors.

```
> let i,j = find sm "a", find sm "d";

> get_elm i, get_elm j;
"a"=>1,"d"=>4

> members (i,j);
["a"=>1,"b"=>2,"c"=>3]

> catch id $ members (j,i); // j and i transposed, C++ would segfault
bad_argument

> erase (sm,"b");  // erase "b"=>2, leaving i and j valid
1

> get_elm i; // still valid
"a"=>1

> erase (sm,"a");  // erase "a"=>1 - invalidating i
1

> catch id $ get_elm i; // bad iterator exception
bad_argument
```

10.9.3 Functions

In this section "acon" always denotes one of the containers that supports interators (stlmap, stlset, stlmmap and stlmset).

`stl::`**`iterator`** i
: Returns a new iterator that points to the same element as `i`.

`stl::`**`begin`** acon
`stl::`**`pastend`** acon
: Returns `acon`'s begin or past-end iterator.

`stl::`**`find`** acon k
: Creates a new iterator that points to an element in `acon` with key equivalent to `k` (if any) or `acon`'s past-end iterator if no such element exists.

`stl::`**`find_with_default`** map k
: Returns an iterator pointing to the element in `map`, a stlmap, with key equivalent to

k. If no such element existed before the call, one is created and inserted using k and map's default value. This function is pure-stlmap's version of C++'s [] operator for associative containers.

stl::**insert_elm** acon elm
: Attempts to insert elm into acon. (If acon is a stlmap or stlmmap, then elm must be a key value pair, (k=>v)). If acon is a stlmap or stlset (i.e., with unique keys) insert_elm returns a pair, the first of which is an iterator pointing to the element with key k that was just inserted (or the pre-existing element that blocked the insertion). The second element in the pair is a boolean value that is true if a new element was inserted. In contrast, if acon is a multi-keyed container (stlmmap or stlmset) the insert will always be successful and insert_elm returns an iterator pointing to the element with key k that was just inserted, instead of an (iterator, boolean) tuple.

stl::**insert_elm** acon (elm,i)
: This is the same as the previous function except that (a) i is passed in as a hint to where the new element should be inserted and (b) a single iterator is returned rather than a iterator,boolean pair. If the new element is inserted just after i, the insertion can have constant time complexity.

stl::**l_bound** acon k
: Return a new iterator that points to the first element in acon, a stlmap, stlset, stlmmap or stlmset, that is not less than k, or acon's past-end iterator if none exists.

stl::**u_bound** acon k
: Return a new iterator that points to the first element in acon, a stlmap, stlset, stlmmap or stlmset, that is greater than k, or acon's past-end iterator if none exists.

stl::**lu_bounds** acon k
: Return the pair l_bound acon k, u_bound acon k.

E.g.,

```
> let ok, smx, f, l = stl::range_info (sm1,"b","e");

> ok, smx === sm1, stl::members (f,l);
1,1,["b"=>2,"c"=>3,"d"=>4]
```

stl::**inc** i
stl::**dec** i
stl::**move** i n::int
: Move the iterator i forward one, back one or forward n elements respectively, where n can be negative. The iterator is mutated by these operations, provided the move is successful. An attempt to move to a position before the first element's position causes an out_of_bounds exception. Moves past the last element return the past-end iterator for the container that i is defined on.

stl::**get_elm** i
stl::**get_key** i
stl::**get_val** i
: Return the element pointed to by the iterator i, or the element's key or value. For maps

the element is returned as a key=>value hash rocket pair. For sets, get_elem, get_key and get_val all return the element (which is the same as its key).

`stl::`**`put_val`** i newvalue
: Change the value of the element pointed to by the iterator `i` to `newvalue`. The element's key cannot be changed. The iterator must point into a stlmap or stlmmap.

`stl::`**`beginp`** i
`stl::`**`pastendp`** i
: Returns true if the iterator `i` is the begin iterator or pastend iterator of the container it is defined on.

`stl::`**`get_info`** i
: Returns a tuple (is_valid,acon,key,val) where is_valid is true if the iterator `i` is valid or false if not, acon is the container that i is defined on, and key, val are the key and value of the element `i` points to, if any. If `i` is the past-end iterator, key and val are set to `stl::smend` and `[]`, respectively.

i **==** j
: Returns true if the iterators `i` and `j` point to the same element.

`erase` (acon,i)
`erase` (acon,i,j)
: Erases the element pointed to by `i` or the elements in the range (`i`, `j`). Both `i` and `j` must be iterators defined on `acon` (or a `bad_argument` exception will be thrown).

10.9.4 Examples

Here are some examples using iterators.

```
> let b,e = begin smm, pastend smm;

> members (b,e);
["a"=>1,"b"=>2,"c"=>31,"c"=>32,"d"=>4,"e"=>5]

> let i,j = lu_bounds smm "c";

> members (b,i);
["a"=>1,"b"=>2]

> members (i,j);
["c"=>31,"c"=>32]

> members (j,e);
["d"=>4,"e"=>5]

> get_elm i;
"c"=>31

> get_elm (inc i);
"c"=>32
```

```
> put_val i 132;
132

> map (\(k=>_)->k=>ord k) (b,i);
["a"=>97,"b"=>98,"c"=>99]

> let is_set, smm1, k, v = get_info i; is_set, members smm1, k, v;
1,["a"=>1,"b"=>2,"c"=>31,"c"=>132,"d"=>4,"e"=>5],"c",132

> get_elm (dec j);
"c"=>132

> inc j $$ inc j $$ get_elm j;
"e"=>5

> inc j $$ endp j;
1
```

10.10 Backward Compatibilty

This section documents changes in pure-stlmap.

10.10.1 pure-stlmap-0.2

Optimized common predicates, such as (<) and (>)

10.10.2 pure-stlmap-0.3

Fixed (>) comparisons on plain old data.

Chapter 11

pure-stlvec

Version 0.4, July 24, 2012

Peter Summerland <p.summerland@gmail.com>

Pure's interface to C++ vectors, specialized to hold pointers to arbitrary Pure expressions, and the C++ Standard Template Library algorithms that act on them.

11.1 Copying

Copyright (c) 2011 by Peter Summerland <p.summerland@gmail.com>.

All rights reserved.

pure-stlvec is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

pure-stlvec is distributed under a BSD-style license, see the COPYING file for details.

11.2 Installation

pure-stlvec-0.4 is included in the "umbrella" addon, *pure-stllib*, which is available at http://code.google.com/p/pure-lang/downloads/list. After you have downloaded and installed *pure-stllib*, you will be able to use pure-stlvec (and *pure-stlmap*, as well).

11.3 Overview

The C++ Standard Template Library ("STL") is a library of generic containers (data structures designed for storing other objects) and a rich set of generic algorithms that operate on them. pure-stlvec provides an interface to one of its most useful containers, "vector", adopted to hold pointers to Pure expressions. The interface provides Pure programmers with a mutable container "stlvec", that, like the STL's vector, holds a sequence of objects that can be accessed in constant time according to their position in the sequence.

11.3.1 Modules

The usual operations for creating, accessing and modifying stlvecs are provided by the stlvec module. Most of the operations are similar in name and function to those provided by the Pure Library for other containers. As is the case for their Pure Library counterparts, these operations are in the global namespace. There are a few operations that have been placed in the stl namespace usually because they do not have Pure Library counterparts.

In addition to the stlvec module, pure-stlvec provides a group of modules, stlvec::modifying, stlvec::nonmodifying, stlvec::sort, stlvec::merge, stlvec::heap, stlvec::minmax and stlvec::numeric, that are straight wrappers the STL algorithms (specialized to work with STL vectors of pointers to Pure expressions). This grouping of the STL algorithms follows that found at http://www.cplusplus.com/reference/algorithm/. This web page contains a table that summarizes of all of the algorithms in one place.

pure-stlvec provides an "umbrella" module, `stlvec::algorithms`, that pulls in all of the STL algorithm interface modules in one go. The STL algorithm wrapper functions reside in the stl namespace and have the same names as their counterparts in the STL.

11.3.2 Simple Examples

Here are some examples that use the basic operations provided by the stlvec module.

```
> using stlvec;

> let sv1 = stlvec (0..4); members sv1;
[0,1,2,3,4]

> insert (sv1,stl::svend) (5..7); members sv1;
STLVEC #<pointer 0xaf4d2c0>
[0,1,2,3,4,5,6,7]

> sv1!3;
3

> sv1!![2,4,6];
[2,4,6]

> replace sv1 3 33; members sv1;
```

```
STLVEC #<pointer 0xaf4d2c0>
[0,1,2,33,4,5,6,7]

> stl::erase (sv1,2,5); members sv1;
STLVEC #<pointer 0xaf4d2c0>
[0,1,5,6,7]

> insert (sv1,2) [2,3,4];  members sv1;
STLVEC #<pointer 0xaf4d2c0>
[0,1,2,3,4,5,6,7]

> let pure_vector = stl::vector (sv1,1,5); pure_vector;
{1,2,3,4}

> stlvec pure_vector;
STLVEC #<pointer 0x9145a38>

> members ans;
[1,2,3,4]

 > map (+10) sv1;
[10,11,12,13,14,15,16,17]

> map (+10) (sv1,2,5);
[12,13,14]

> foldl (+) 0 sv1;
28

> [x+10 | x = sv1; x mod 2];
[11,13,15,17]

> {x+10 | x = (sv1,2,6); x mod 2};
{13,15}
```

Here are some examples that use STL algorithms.

```
> using stlvec::algorithms;

> stl::reverse (sv1,2,6); members sv1;
()
[0,1,5,4,3,2,6,7]

> stl::stable_sort sv1 (>); members sv1;
()
[7,6,5,4,3,2,1,0]

> stl::random_shuffle sv1; members sv1 1;
()
[1,3,5,4,0,7,6,2]

> stl::partition sv1 (<3); members (sv1,0,ans); members sv1;
```

```
3
[1,2,0]
[1,2,0,4,5,7,6,3]

> stl::transform sv1 (sv1,0) (*2); members sv1;
-1
[2,4,0,8,10,14,12,6]

> let sv2 = emptystlvec;

> stl::transform sv1 (sv2,stl::svback) (div 2); members sv2;
-1
[1,2,0,4,5,7,6,3]
```

Many more examples can be found in the pure-stlvec/ut directory.

11.3.3 Members and Sequences of Members

Throughout the documentation for pure-stlvec, the member of a stlvec that is at the nth position in the sequence of expressions stored in the stlvec is referred to as its nth member or nth element. The nth member of a stlvec, sv, is sometimes denoted by sv!n. The sequence of members of sv starting at position i up to but not including j is denoted by sv[i,j). There is a "past-the-end" symbol, stl::svend, that denotes the position after that occupied by the last member contained by a stlvec.

For example, if sv contains the sequence "a", "b", "c" "d" and "e", sv!0 is "a", sv[1,3) is the sequence consisting of "b" followed by "c" and v[3,stl::svend) denotes the sequence consisting of "d" followed by "e".

11.3.4 STL Iterators and Value Semantics

In C++ a programmer accesses a STL container's elements by means of "iterators", which can be thought of as pointers to the container's elements. A single iterator can be used to access a specific element, and a pair of iterators can be used to access a "range" of elements. By convention, such a range includes the member pointed to by the first iterator and all succeeding members up to but not including the member pointed to by the second iterator. Each container has a past-the-end iterator that can be used to specifiy ranges that include the container's last member.

In the case of vectors there is an obvious correspondence between an iterator that points to an element and the element's position (starting at zero) in the vector. pure-stlvec uses this correspondence to designate a stlvec's members in a way that makes it relatively easy to see how pure-stlvec's functions are acting on the stlvec's underlying STL vector by referencing the STL's documentation. Thus, if sv is a stlvec, and j is an int, "replace sv j x" uses the STL to replace the element pointed to by the iterator for position j of sv's underlying STL vector. If, in addition, k is an int, stl::sort (sv,j,k) (<) uses the STL to sort the elements in the range designated by the "jth" and "kth" iterators for sv's underlying STL vector. This range,

written as sv[j,k), is the subsequence of sv that begins with the element at position j and ends with the element at position (k-1).

Besides iterators, another cornerstone of the STL is its "value semantics", i.e., all of the STL containers are mutable and if a container is copied, all of its elements are copied. pure-stlvec deals with the STL's value semantics by introducing mutable and nonmutable stlvecs, and by storing smart pointers to objects (which have cheap copies) rather than the actual objects.

11.3.5 Iterator Tuples

As mentioned in the previous section, in C++ ranges are specified by a pair of STL iterators.

In pure-stlvec ranges of elements in a stlvec are specified by "iterator tuples" rather than, say, actual pointers to STL iterators. Iterator tuples consist of the name of a stlvec followed by one of more ints that indicate positions (starting from zero) of the stlvec's elements.

To illustrate how iterator tuples are used, consider the STL stable_sort function, which sorts objects in the range [first, last) in the order imposed by comp. Its C++ signature looks like this:

> void stable_sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp)

The corresponding pure-stlvec function, from the stlvec::sort module, looks like this:

```
stable_sort (msv, first, last) comp
```

where msv is a mutable stlvec, and first and last are ints. The first thing that the Pure stable_sort does is create a pair of C++ iterators that point to the elements in msv's underlying STL vector that occupy the positions designated by first and last. Next it wraps the Pure comp function in a C++ function object that, along with the two iterators, is passed to the C++ stable_sort function.

For convenience, (sv,stl::svbeg, stl::svend) can be written simply as sv. Thus, if first were stl::svbeg (or 0), and last were stl::svend (or #msv, the number of elements in msv), the last Pure call could be written:

```
stable_sort msv comp
```

It should be noted that often the STL library provides a default version of its functions, which like stable_sort, use a comparator or other callback function provided by the caller. E.g., the C++ stable_sort has a default version that assumes the "<" operator can be used on the elements held by the container in question:

> void stable_sort (RandomAccessIterator first, RandomAccessIterator last)

The corresponding functions provided by the pure-stlvec modules rarely, if ever, supply a default version. A typical example is stlvec::sort's stable_sort which must be called with a comparator callback function:

```
stable_sort msv (<);
```

Note also that the comparator (e.g., (<)), or other function being passed to a pure-stlvec algorithm wrapper is almost always the last parameter. This is the opposite of what is required for similar Pure functions, but is consistent with the STL calling conventions.

11.3.6 Predefined Iterator Tuple Indexes

The following integer constants are defined in the stl namespace for use in iterator tuples.

constant `stl::`**`svbeg`** = 0
constant `stl::`**`svend`** = -1
constant `stl::`**`svback`** = -2

These three symbols are declared as nonfix. `svend` corresponds to STL's past-end iterator for STL vectors. It makes it possible to specify ranges that include the last element of an stlvec. I.e., the iterator tuple (sv,stl::svbeg,stl::svend) would specify sv[0,n), where n is the number of elements in sv. In order to understand the purpose of `svback`, it is necessary to understand a bit about STL's "back insert iterators."

11.3.7 Back Insert Iterators

Many of the STL algorithms insert members into a target range designated by an iterator that points to the first member of the target range. Consistent with raw C usage, it is ok to copy over existing elements the target stlvec. E.g.,:

```
> using stlvec::modifying;

> let v1 = stlvec (0..2);

> let v2 = stlvec ("a".."g");

> stl::copy v1 (v2,2) $$ members v2;
["a","b",0,1,2,"f","g"]
```

This is great for C++ programmers, but for Pure programmers it is almost always preferable to append the copied items to the end of a target stlvec, rather than overwriting all or part or part of it. This can be accomplished using stl::svback. E.g.,:

```
> stl::copy v1 (v2,stl::svback) $$ members v2;
["a","b",0,1,2,"f","g",0,1,2]
```

In short, when a pure-stlvec function detects "stl::svback" in a target iterator tuple, it constructs a STL "back inserter iterator" and passes it on to the corresponding wrapped STL function.

11.3.8 Data Structure

Currently, stlvecs are of the form (STLVEC x) or (CONST_STLVEC x), where STLVEC AND CONST_STLVEC are defined as nonfix symbols in the global namespace and x is a pointer

to the underlying STL vector. The stlvec module defines corresponding type tags, stlvec and const_stlvec, so the programmer never needs to worry about the underlying representaton.

This representation may change in the future, and must not be relied upon by client modules. In particular, one must never attempt to use the embedded pointer directly.

As the names suggest, stlvecs are mutable and const_stlvecs are immutable. Functions that modify a stlvec will simply fail unless the stlvec is mutable.

```
> let v = const_stlvec $ stlvec (0..3); v2;
CONST_STLVEC #<pointer 0x8c1dbf0>

> replace v 0 100; // fails
replace (CONST_STLVEC #<pointer 0x9f07690> 0 100
```

11.3.9 Types

pure-stlvec introduces six type tags, all of which are in the global namespace:

type `mutable_stlvec`
: The type for a mutable stlvec.

type `const_stlvec`
: The type for an immutable stlvec.

type `stlvec`
: The type for a stlvec, mutable or immutable.

type `mutable_svit`
: The type for an iterator tuple whose underlying stlvec is mutable.

type `const_svit`
: The type for an iterator tuple whose underlying stlvec is immutable.

type `svit`
: The type for an iterator tuple. The underlying stlvec can be mutable or immutable.

11.3.10 Copy-On-Write Semantics

The pure-stlvec module functions do not implement automatic copy-on-write semantics. Functions that modify stlvec parameters will simply fail if they are passed a const_stlvec when they expect a mutable_stlvec.

For those that prefer immutable data structures, stlvecs can be converted to const_stlvecs (usually after they have been created and modified within a function) by the `const_stlvec` function. This function converts a mutable stlvec to an immutable stlvec without changing the underlying STL vector.

Typically, a "pure" function that "modifies" a stlvec passed to it as an argument will first copy the input stlvec to a new locally scoped (mutable) stlvec using the stlvec function. It

will then modify the new stlvec and use const_stlvec to make the new stlvec immutable before it is returned. It should be noted that several of the STL algorithms have "copy" versions which place their results directly into a new stlvec, which can eliminate the need to copy the input stlvec. E.g.:

```
> let sv1 = stlvec ("a".."e");

> let sv2 = emptystlvec;

> stl::reverse_copy sv1 (sv2,stl::svback) $$ members sv2;
["e","d","c","b","a"]
```

Without reverse_copy, one would have had to copy sv1 into sv2 and then reverse sv2.

If desired, in Pure it is easy to write functions that have automatic copy-on-write semantics. E.g.,

```
> my_replace csv::const_stlvec i x = my_replace (stlvec csv) i x;
> my_replace sv::stlvec i x = replace sv i x;
```

11.3.11 Documentation

The pure-stllib/doc directory includes a rudimentary cheatsheet, pure-stllib-cheatsheet.pdf, that shows the signatures of all of the functions provided by pure-stlvec (and by *pure-stlmap* as well).

The documentation of the functions provided by the stlvec module are reasonably complete. In contrast, the descriptions of functions provided by the STL algorithm modules are purposely simplified (and may not, therefore, be technically accurate). This reflects that fact that the functions provided by pure-stlvec have an obvious correspondence to the functions provided by the STL, and the STL is extremely well documented. Furthermore, using the Pure interpreter, it is very easy to simply play around with with any of the pure-stlvec functions if there are doubts, especially with respect to "corner cases." Often this leads to a deeper understanding compared to reading a precise technical description.

A good book on the STL is STL Tutorial and Reference Guide, Second Edition, by David R. Musser, Gillmer J. Derge and Atul Saini. A summary of all of the STL algorithms can be found at http://www.cplusplus.com/reference/stl/.

11.3.12 Parameter Names

In the descriptions of functions that follow, parameter names used in function descriptions represent specific types of Pure objects:

sv stlvec (mutable or immutable)

csv const (i.e., immutable) stlvec

msv mutable stlvec

x an arbitrary Pure expression

xs a list of arbitrary Pure expressions

count, sz, n whole numbers to indicate a number of elements, size of a vector, etc

i,j whole numbers used to designate indexes into a stlvec

f,m,l whole numbers (or stl::beg or stl::svend) designating the "first", "middle" or "last" iterators in a stlvec iterator tuple

p a whole number (or other iterator constant such as stl::svend or stl::svback) used in a two element iterator tuple (e.g., (sv,p))

(sv,p) an iterator tuple that will be mapped to an iterator that points to the pth position of sv's underlying STL vector, v, (or to a back iterator on v if p is stl::svback)

(sv,f,l) an iterator tuple that will be mapped to the pair of iterators that are designated by (sv,f) and (sv,l)

(sv,f,m,l) an iterator tuple that will be mapped to the iterators that are designated by (sv,f), (sv,m) and (sv,l)

sv[f,l) the range of members beginning with that at (sv,f) up to but not including that at (con,l)

comp a function that accepts two objects and returns true if the first argument is less than the second (in the strict weak ordering defined by comp), and false otherwise

unary_pred a function that accepts one object and returns true or false

bin_pred a function that accepts two objects and returns true or false

unary_fun a function that accepts one objects and returns another

bin_fun a function that accepts two objects and returns another

gen_fun a function of one parameter that produces a sequence of objects, one for each call

For readability, and to correspond with the STL documentation, the words "first", "middle", and "last", or variants such as "first1" are often used instead of f,m,l.

11.4 Error Handling

The functions provided this module handle errors by throwing exceptions.

11.4.1 Exception Symbols

constructor `bad_argument`
: This exception is thrown when a function is passed an unexpected value. A subtle error to watch for is a malformed iterator tuple (e.g., one with the wrong number of elements).

constructor `bad_function`
: This exception is thrown when a purported Pure call-back function is not even callable.

constructor `failed_cond`
: This exception is thrown when a Pure call-back predicate returns a value that is not an int.

constructor `out_of_bounds`
: This exception is thrown if the specified index is out of bounds.

constructor `range_overflow`
: This exception is thrown by functions that write over part of a target stlvec (e.g., copy) when the target range too small to accommodate the result.

constructor `range_overlap`
: This exception is thrown by algorithm functions that write over part of a target stlvec when the target and source ranges overlap in a way that is not allowed.

In addition, any exception thrown by a Pure callback function passed to a pure-stlvec function will be caught and be rethrown by the pure-stlvec function.

11.4.2 Examples

```
> using stlvec, stlvec::modifying;

> let sv1 = stlvec (0..4); members sv1;
[0,1,2,3,4]

> let sv2 = stlvec ("a".."e"); members sv2;
["a","b","c","d","e"]

> sv1!10;
<stdin>, line 25: unhandled exception 'out_of_bounds' ...

> stl::copy sv1 (sv2,10);
<stdin>, line 26: unhandled exception 'out_of_bounds' ...

> stl::copy sv1 (sv2,2,3); // sb (sv2,pos)
<stdin>, line 22: unhandled exception 'bad_argument' ...

> stl::copy sv1 (sv2,2);
<stdin>, line 23: unhandled exception 'range_overflow' ...

> stl::copy sv2 (sv2,2);
<stdin>, line 24: unhandled exception 'range_overlap' ...

> stl::copy (sv1,1,3) (sv2,0); members sv2; // ok
2
[1,2,"c","d","e"]

> stl::sort sv2 (>); // apples and oranges
```

```
<stdin>, line 31: unhandled exception 'failed_cond'

> listmap (\x->throw DOA) sv1; // callback function throws exception
<stdin>, line 34: unhandled exception 'DOA' ...
```

11.5 Operations Included in the stlvec Module

The stlvec module provides functions for creating, accessing and modifying stlvecs. In general, operations that have the same name as a corresponding function in the Pure standard library are in the global namespace. The remaining functions, which are usually specific to stlvecs, are in the stl namespace.

Please note that "stlvec to stlvec" functions are provided by the pure-stl algorithm modules. Thus, for example, the stlvec module does not provide a function that maps one stlvec onto a new stlvec. That functionality, and more, is provided by stl::transform, which can be found in the stlvec::modifying module.

11.5.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec;
```

11.5.2 Operations in the Global Namespace

When reading the function descriptions that follow, please bear in mind that whenever a function is passed an iterator tuple of the form (sv,first, last), first and last can be dropped, leaving (sv), or simply sv. The function will treat the "unary" iterator tuple (sv) as (sv, stl::svbeg, stl::svend).

emptystlvec
: return an empty stlvec

stlvec source /stlvec
: create a new stlvec that contains the elements of source; source can be a stlvec, an iterator tuple(sv,first,last), a list or a vector (i.e., a matrix consisting of a single row or column). The underlying STL vector is always a new STL vector. I.e., if source is a stlvec the new stlvec does not share source's underlying STL vector.

mkstlvec x count
: create a new stlvec consisting of count x's.

const_stlvec source
: create a new const_stlvec that contains the elements of source; source can be a stlvec, an iterator tuple(sv,first,last), a list or a vector (i.e., a matrix consisting of a single row

or column). If source is a stlvec (mutable or const), the new const_stlvec shares source's underlying STL vector.

sv
: return the number of elements in sv.

Note that # applied to an iterator tuple like (sv,b,e) will just return the number of elements in the tuple. Use stl::bounds if you need to know the number of elements in the range denoted by an iterator tuple.

sv **!** i
: return the ith member of sv

Note that !k applied to an iterator tuple like (sv,b,e) will just return the kth element of the tuple. In addition, in stlvec, integers used to denote postions (as in !k) or in iterators, *always*, are relative to the beginning of the underlying vector. So it makes no sense to apply ! to an iterator tuple.

first sv
last sv
: first and last member of sv

members (sv, first, last)
: return a list of values stored in sv[first,last)

replace msv i x
: replace the ith member of msv by x and return x; throws out_of_bounds if i is less than 0 or great or equal to the number of elements in msv

update msv i x
: the same as replace except that update returns msv instead of x. This function is DEPRECATED.

append sv x
: append x to the end of sv

insert (msv,p) xs
insert (msv,p) (sv,first,last)
: insert members of the list xs or the range sv[first, last) into msv, all preceding the pth member of msv. Members are shifted to make room for the inserted members

rmfirst msv
rmlast msv
: remove the first or last member from msv

erase (msv,first,last)
erase (msv,p)
erase msv
: remove msv[first,last) from msv, remove msv!p from msv, or make msv empty. Members are shifted to occupy vacated slots

sv1 **==** sv2

sv1 **~=** sv2
: (x == y) is the same as stl::allpairs (==) x y and x ~= y is simply ~(allpairs (==) x y)

Note that == and ~== are not defined for iterator tuples (the rules would never be executed because == is defined on tuples in the Prelude).

The stlvec module provides convenience functions that apply map, catmap, foldl, etc, to directly access Pure expressions stored in a stlvec.

map unary_fun (sv, first, last)
: one pass equivalent of map unary_fun $ members (sv, first, last)

listmap unary_fun (sv, first, last)
: same as map, used in list comprehensions

catmap unary_fun (sv, first, last)
: one pass equivalent of catmap unary_fun $ members (sv, first, last)

do unary_fun (sv, first, last)
: one pass equivalent of do unary_fun $ members (sv, first, last)

foldl bin_fun x (sv, first, last)
: one pass equivalent of foldl bin_fun x $ members (sv, first, last)

foldl1 bin_fun (sv, first, last)
: one pass equivalent of foldl1 bin_fun $ members (sv, first, last)

filter unary_pred (sv, first, last)
: one pass equivalent of filter unary_pred $ members (sv, first, last)

The following four functions map (or catmap) stlvecs onto row and col matrixes, primarily for use in matrix comprehensions.

rowmap unary_fun (sv, first, last)

rowcatmap unary_fun (sv, first, last)

colmap unary_fun (sv, first, last)

colcatmap unary_fun (sv, first, last)

11.5.3 Operations in the stl Namespace

stl::**empty** sv
: test whether sv is empty

stl::**vector** (sv,first,last)
: create a Pure vector that contains the members of sv[first,last)

stl::**allpairs** bin_pred (sv1, first1, last1) (sv2, first2, last2)
: returns true if bin_pred is true for all corresponding members of sv1[first1, last1) and sv2[first2, last2)

`stl::`**`bounds`** (sv,first,last)
: throws out-of-bounds if first or last is out of bounds. returns the tuple (sv,first,last) except that if first is stl::begin it will be replaced by 0 and if last is stl::svend it will be replaced by the number of elements in sv.

`stl::`**`reserve`** msv count
: modify the underlying STL vector to have at least count slots, useful for packing data into a fixed size vector and possibly to speed up the addition of new members

`stl::`**`capacity`** sv
: return the number of slots (as opposed to the number of elements) held by the underlying STL vector

11.5.4 Examples

See ut_stlvec.pure and ut_global_stlvec.pure in the pure-stlvec/ut directory.

11.6 STL Nonmodifying Algorithms

The stlvec::nonmodifying module provides an interface to the STL's non-modifying sequence operations.

11.6.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::nonmodifying;
```

All of the functions are in the stl namespace.

11.6.2 Operations

`stl::`**`for_each`** (sv, first, last) unary_fun
: applies unary_fun to each of the elements in sv[first,last)

`stl::`**`find`** (sv, first, last) x
: returns the position of the first element in sv[first,last) for which (==x) is true (or stl::svend if not found)

`stl::`**`find_if`** (sv, first, last) unary_pred
: returns the position of the first element in sv[first,last) for which unary_pred is true (or stl::svend if not found)

`stl::`**`find_first_of`** (sv1, first1, last1) (sv2, first2, last2) bin_pred
: Returns the position of the first element, x, in sv1[first1,last1) for which there exists y in sv2[first2,last2) and (bin_pred x y) is true (or stl::svend if no such x exists).

`stl::`**`adjacent_find`** (sv, first, last) bin_pred
: search sv[first,last) for the first occurrence of two consecutive elements (x,y) for which (bin_pred x y) is true. Returns the position of x, if found, or stl::svend if not found)

`stl::`**`count`** (sv, first, last) x
: returns the number of elements in the range sv[first,last) for which (x==) is true

`stl::`**`count_if`** (sv, first, last) unary_pred
: returns the number of elements in the range sv[first,last) for which unary_pred is true

`stl::`**`mismatch`** (sv1, first1, last1) (sv2, first2) bin_pred
: applies bin_pred pairwise to the elements of sv1[first1,last1) and (sv2,first2,first2 + n), with n equal to last1-first1 until it finds i and j such that bin_pred (sv1!i) (sv2!j) is false and returns (i,j). If bin_pred is true for all of the pairs of elements, i will be stl::svend and j will be first2 + n (or stl::svend)

`stl::`**`equal`** (sv1, first1, last1) (sv2, first2) bin_pred
: applies bin_pred pairwise to the elements of sv1[first1,last1) and (sv2,first2,first2 + n), with n equal to last1-first1, and returns true if bin_pred is true for each pair

`stl::`**`search`** (sv1, first1, last1) (sv2, first2) bin_pred
: using bin_pred to determine equality of the elements, searches sv1[first1,last1) for the first occurrence of the sequence defined by sv2[first2,last2), and returns the position in sv1 of its first element (or stl::svend if not found)

`stl::`**`search_n`** (sv, first, last) count x bin_pred
: using bin_pred to determine equality of the elements, searches sv[first,last) for a sequence of count elements that equal x. If such a sequence is found, it returns the position of the first of its elements, otherwise it returns stl::svend

`stl::`**`find_end`** (sv1, first1, last1) (sv2, first2, last2) bin_pred
: using bin_pred to determine equality of the elements, searches sv1[first1,last1) for the last occurrence of sv2[first2,last2). Returns the position of the first element in sv1 of the occurrence (or stl::svend if not found).

11.6.3 Examples

See ut_nonmodifying.pure in the pure-stlvec/ut directory.

11.7 STL Modifying Algorithms

The stlvec::modifying module provides an interface to the STL's modifying algorithms.

11.7.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::modifying;
```

All of the functions are in the stl namespace.

11.7.2 Operations

`stl::`**`copy`** (sv, first1, last1) (msv, first2)
: copies the elements in sv[first1,last1) into the range whose first element is (msv,first2)

`stl::`**`copy_backward`** (sv,first1,last1) (msv,last2)
: copies the elements in sv[first1,last1), moving backward from (last1), into the range msv[first2,last2) where first2 is last2 minus the number of elements in sv[first1,last1)

`stl::`**`swap_ranges`** (sv,first,last) (msv, p)
: exchanges the elements in sv[first, last) with those in msv[p, p+n) where n is last - first

`stl::`**`transform`** (sv,first,last) (msv, p) unary_fun
: applies unary_fun to the elements of sv[first,last) and places the resulting sequence in msv[p, p+n) where n is last - first. If sv is mutable, msv and sv can be the same stlvec. Returns (msv,p+n)

`stl::`**`transform_2`** (sv1,first1,last1) (sv2,first2) (msv, p) bin_fun
: applies bin_fun to corresponding pairs of elements of sv1[first1,last1) sv2[first2,n) and and places the resulting sequence in msv[p, p+n) where n is last1 - first1. Returns (msv,p+n)

`stl::`**`replace_if`** (msv,first,last) unary_pred x
: replace the elements of msv[first,last) that satistfy unary_pred with x

`stl::`**`replace_copy`** (sv,first,last) (msv,p) x y
: same as `replace` (msv,first,last) x y except that the modified sequence is placed in msv[p,p+last-first)

`stl::`**`replace_copy_if`** (sv,first,last) (msv,p) unary_pred x
: same as `replace_if` except that the modified sequence is placed in msv[p,p+last-first)

`stl::`**`fill`** (msv,first,last) x
: replace all elements in msv[first,last) with x

`stl::`**`fill_n`** (msv,first) n x
: replace the elements of msv[first,first+n) with x

`stl::`**`generate`** (msv,first,last) gen_fun
: replace the elements in msv[first,last) with the sequence generated by successive calls to gen_fun (), e.g.,

```
> let count = ref 0;

> g _ = n when n = get count + 1; put count n; end;

> let sv = mkstlvec 0 10;
```

```
> stl::generate sv g $$ members sv;
[1,2,3,4,5,6,7,8,9,10]
```

`stl::`**`generate_n`** (msv,first) n gen_fun
: replace all elements in msv[first,first+n) with the sequence generated by successive calls to gen_fen

`stl::`**`remove`** (msv,first,last) x
: same as `remove_if` (msv,first,last) (==x).

`stl::`**`remove_if`** (msv,first,last) unary_pred
: remove elements in msv[first,last) that satisfy unary_pred. If n elements do not satisfy unary_pred, they are moved to msv[first,first+n), preserving their relative order. The content of msv[first+n,svend) is undefined. Returns first+n, or stl::svend if first+n is greater than the number of elements in msv

`stl::`**`remove_copy`** (sv,first,last) (msv,first) x
: same as `remove` except that the purged sequence is copied to (msv,first) and sv[first,last) is not changed

`stl::`**`remove_copy_if`** (sv,first,last) (msv,first) unary_pred
: same as `remove_if` except that the purged sequence is copied to (msv,first) and sv[first,last) is not changed

`stl::`**`unique`** (msv,first,last) bin_pred
: eliminates consecutive duplicates from sv[first,last), using bin_pred to test for equality. The purged sequence is moved to sv[first,first+n) preserving their relative order, where n is the size of the purged sequence. Returns first+n or stl::svend if first+n is greater than the number of elements in msv

`stl::`**`unique_copy`** (sv,first,last) (msv,first) bin_pred
: same as `unique` except that the purged sequence is copied to (msv,first) and sv[first,last) is not changed

`stl::`**`reverse`** (msv,first,last)
: Reverses the order of the elements in sv[first,last).

`stl::`**`reverse_copy`** (sv,first,last) (msv,first)
: same as `reverse` except that the reversed sequence is copied to (msv,first) and sv[first,last) is not changed.

`stl::`**`rotate`** (msv,first,middle,last)
: rotates the elements of msv[first,middle,last] so that middle becomes the first element of msv[first,last].

`stl::`**`rotate_copy`** (msv,first,middle,last) (msv,first)
: same as rotate except that the rotated sequence is copied to (msv,first) and sv[first,last) is not changed.

`stl::`**`random_shuffle`** (msv,first,last) int::seed
: randomly reorders the elements in msv[first,last)

`stl::`**`partition`** (msv,first,last) unary_pred
: places the elements in msv[first,last) that satisfy unary_pred before those that don't. Returns middle, where msv [first,middle) contains all of the elements that satisfy unary_pre, and msv [middle, last) contains those that do not

`stl::`**`stable_partition`** (msv,first,last) unary_pred
: same as partition except that the relative positions of the elements in each group are preserved

11.7.3 Examples

See ut_modifying.pure in the pure-stlvec/ut directory.

11.8 STL Sort Algorithms

The stlvec::sort module provides an interface to the STL's sorting and binary search algorithms.

11.8.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::sort;
```

All of the functions are in the stl namespace.

11.8.2 Operations

All of the functions in this module require the caller to supply an ordering function, comp. The functions (<) and (>) are commonly passed as comp.

`stl::`**`sort`** (msv, first, last) comp
: sorts msv[first, last)

`stl::`**`stable_sort`** (msv, first, last) comp
: sorts msv[first, last), preserving the relative order of equal members

`stl::`**`partial_sort`** (msv, first, middle, last) comp
: fills msv[first, middle) with the elements of msv[first,last) that would appear there if msv[first,last) were sorted using comp and fills msv[middle,last) with the remaining elements in unspecified order

`stl::`**`partial_sort_copy`** (sv, first1, last1) (msv, first2, last2) comp
: let n be the number of elements in sv[first1, last1) and r be the number of elements in msv[first2, last2). If r < n, `partial_sort_copy` fills msv[first2, last2) with the first r elements of what sv[first1, last1) would be if it had been sorted. If r >= n, it fills

msv[first2, first2+n) with the elements of sv[first1, last1) in sorted order. sv[first1,last1) is unchanged

`stl::`**`nth_element`** (msv, first, middle, last) comp
: rearranges the elements of msv[first, last) as follows. Let n be middle - first, and let x be the nth smallest element of msv[first, last). After the function is called, sv!middle will be x. All of the elements of msv[first, middle) will be less than x and all of the elements of msv[middle+1, last) will be greater than x

The next four functions assume that sv[first, last) is ordered by comp.

`stl::`**`lower_bound`** (sv, first, last) x comp
: returns an int designating the first position into which x can be inserted into sv[first, last) while maintaining the sorted ordering

`stl::`**`upper_bound`** (sv, first, last) x comp
: returns an int designating the last position into which x can be inserted into sv[first, last) while maintaining the sorted ordering

`stl::`**`equal_range`** (sv, first, last) x comp
: returns a pair of ints, (lower, upper) where lower and upper would have been returned by separate calls to lower_bound and upper_bound.

`stl::`**`binary_search`** (sv, first, last) x comp
: returns true if x is an element of sv[first, last)

11.8.3 Examples

See ut_sort.pure in the pure-stlvec/ut directory.

11.9 STL Merge Algorithms

The stlvec::merge module provides an interface to the STL's merge algorithms. These algorithms operate on sorted ranges.

11.9.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::merge;
```

All of the functions are in the stl namespace.

11.9.2 Operations

All of the functions in this module require the caller to supply an ordering function, comp (as for the Pure library sort function). They only work properly on input ranges that have been previously sorted using comp. The set operations generally do not check for range overflow because it is not generally possible to determine the length of the result of a set operation until after it is completed. In most cases you will get a nasty segmentation fault if the result is bigger than the target range. The best way to avoid this possibility it to use a back iterator to specifify the target range.

See parameter naming conventions at ..

`stl::`**`merge`** (sv1,first1,last1) (sv2,first2,last2) (msv,p) comp
: merges the two sorted ranges into the sorted range msv[p,p+n) where n is the total length of the merged sequence

`stl::`**`inplace_merge`** (msv,first, middle, last) comp
: merges msv[first,middle) and msv[middle,last) into the sorted range msv[first,last)

`stl::`**`includes`** (sv1,first1,last1) (sv2,first2,last2) comp
: returns true if every element of sv2[first2,last2) is an element of sv1[first1,last1)

`stl::`**`set_union`** (sv1,first1,last1) (sv2,first2,last2) (msv,p) comp
: places the sorted union of sv1[first1,last1) and sv2[first2,last2) into msv[p,p+n) where n is the number of elements in the sorted union, and returns the past-the-end position of the sorted union

`stl::`**`set_intersection`** (sv1,first1,last1) (sv2,first2,last2) (msv,p) comp
: places the sorted intersection of sv1[first1,last1) and sv2[first2,last2) into msv[p,p+n) where n is the number of elements in the sorted intersection, and returns p+n (or stl::svend, if applicable)

`stl::`**`set_difference`** (sv1,first1,last1) (sv2,first2,last2) (msv,p) comp
: places the sorted difference of sv1[first1,last1) and sv2[first2,last2) into msv[p,p+n) where n is the number of elements in the sorted difference, and returns p+n (or stl::svend, if applicable)

`stl::`**`set_symmetric_difference`** (sv1,first1,last1) (sv2,first2,last2) (msv,p) comp
: places the sorted symmetric_difference of sv1[first1,last1) and sv2[first2,last2) into msv[p,p+n) where n is the number of elements in the sorted symmetric_difference, and returns returns p+n (or stl::svend, if applicable)

11.9.3 Examples

See ut_merge.pure in the pure-stlvec/ut directory.

11.10 STL Heap Algorithms

The stlvec::heap module provides an interface to the STL's heap operations.

11.10.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::heap;
```

All of the functions are in the stl namespace.

11.10.2 Operations

All of the functions in this module require the caller to supply an ordering function, comp (as for the Pure library sort function). The functions (<) and (>) are commonly passed as comp.

`stl::`**`make_heap`** (msv,first,last) comp
: rearranges the elements of msv[first,last) so that they are a heap, i.e., after this msv!first will be the largest element in msv[first,last), and push_heap and pop_heap will work properly

`stl::`**`push_heap`** (msv,first,last) comp
: makes msv[first,last) a heap (assuming that msv[first,last-1) was a heap)

`stl::`**`pop_heap`** (msv,first,last) comp
: swaps msv!first with msv!(last-1), and makes msv[first,last-1) a heap (assuming that msv[first,last) was a heap)

`stl::`**`sort_heap`** (msv,first,last) comp
: sorts the elements in msv[first,last)

11.10.3 Examples

See ut_heap.pure in the pure-stlvec/ut directory.

11.11 Min/Max STL Algorithms

The stlvec::minmax module provides an interface to a few additional STL algorithms.

11.11.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::minmax;
```

All of the functions are in the stl namespace.

11.11.2 Operations

All of the functions in this module require the caller to supply an ordering function, comp (as for the Pure library sort function). The functions (<) and (>) are commonly passed as comp.

`stl::`**`min_element`** (sv,first,last) comp
: returns the position of the minimal element of sv[first,last) under the ordering defined by comp

`stl::`**`max_element`** (sv,first,last) comp
: returns the position of the maximal element of sv[first,last) under the ordering defined by comp

`stl::`**`lexicographical_compare`** (sv1,first1,last1) (sv2,first2,last2) comp
: compares sv1[first1,last1) and sv2[first2,last2) element by element according to the ordering defined by comp, and returns true if the first sequence is less than the second

Algorithms are provided for stepping through all the permutations the elements of a stlvec. For these purposes, the first permutation has the elements of msv[first,last) sorted in ascending order and the last has the elements sorted in descending order.

`stl::`**`next_permutation`** (msv,first,last) comp
: rearranges msv[first,last) to produce the next permutation, in the ordering imposed by comp. If the elements of the next permutation is ordered (ascending or decending) by comp, return false. Otherwise return true.

`stl::`**`prev_permutation`** (msv,first,last) comp
: next_permutation in reverse

11.11.3 Examples

See ut_minmax.pure in the pure-stlvec/ut directory.

11.12 STL Numeric Algorithms

The stlvec::numeric module provides an interface to the STL's numeric algorithms.

11.12.1 Imports

To use the operations of this module, add the following import declaration to your program:

```
using stlvec::numeric;
```

All of the functions are in the stl namespace.

11.12.2 Operations

`stl::`**`accumulate`** (sv,first,last) x bin_fun
: accumulate bin_fun over x and the members of sv[first,last), like foldl

`stl::`**`inner_product`** (sv1,first1,last1) (sv2,first2,last2) x bin_fun1 bin_fun2
: initialize ret with x. Traverse pairs of elements of sv1[first1,last1) and sv2[first2,last2), denoted by (e1, e2), replacing ret with (bin_fun1 ret $ bin_fun2 e1 e2). The number pairs traversed is equal to the size of sv1[first1,last1)

`stl::`**`partial_sum`** (sv,first,last) (msv, p) bin_fun
: accumulate bin_fun f over the elements of sv1[first1,last1), placing itermediate results in msv[p,p+n), where n is last - first, and returns q where m is q - n and msv[m,q) is the intermediate sequence

`stl::`**`adjacent_difference`** (sv,first,last) (msv, p) bin_fun
: produce a sequence of new elements by applying bin_fun to adjacent elements of sv[first,last), placing the new elements in msv[p,p+n), where n is last - first, with the intermediate results, and returns q where m is q - n and msv[m,q) is the new sequence

11.12.3 Examples

See ut_numeric.pure in the pure-stlvec/ut directory.

11.13 Reference Counting

The following function, also in the stl namespace, is available if you want to observe how pure-stlvec maintains reference counts for items in its containers.

`stl::`**`refc`** x
: returns the x's reference count (maintained by the Pure runtime for garbage collection purposes)

11.14 Backward Compatibilty

This section documents changes in pure-stlvec that might have introduced backward compatiblity issues.

11.14.1 pure-stlvec-0.2

Bug fixes.

11.14.2 pure-stlvec-0.3

Version 0.3 reflects some changes made to make *pure-stlvec* consistent with its sister package, *pure-stlmap*.

The `update` function was deprecated. Please use `replace` instead.

The `replace` function was added to the stlvec module. This function is the same as `update` except that "`replace` sv i x" returns x instead of sv.

The `stl::replace` function was removed from the stlvec/modifying module. You can use "`stl::replace_if` (sv,first,last) (x==) y" instead of "`stl::replace` (sv,first,last) x y" to replace all instances of x in the specified range.

The function `null` was removed and `stl::empty` was added to replace it.

The function `list` was removed. You can use `members` instead.

The function `stl::random_shuffle` was changed to take a seed as a second parameter.

All of the tracing functions were removed.

11.14.3 pure-stlvec-0.4

Fixed (>) predicate operating on plain old data when passed to STL algorithms.

Chapter 12

Gnumeric/Pure: A Pure Plugin for Gnumeric

Version 0.12, June 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

Gnumeric/Pure is a Gnumeric extension which lets you use Pure functions in Gnumeric, the Gnome spreadsheet. It offers better execution speed than the existing Perl and Python plugins, and provides some powerful features not found in other Gnumeric scripting plugins, such as asynchronous data sources created from Pure streams and OpenGL rendering in Gnumeric frame widgets via Pure's OpenGL module.

12.1 Introduction

This package provides a Gnumeric extension which gives you access to the Pure programming language in Gnumeric. It works pretty much like the Perl and Python plugin loaders which are distributed with Gnumeric, but Gnumeric/Pure offers some powerful features which aren't found in other Gnumeric scripting plugins:

- Pure is a functional programming language which fits the computational model of spreadsheet programs very well.
- Pure is based on term rewriting and thus enables you to do symbolic computations in addition to the usual numeric calculations.
- Pure has a built-in MATLAB/Octave-like matrix data structure which makes it easy to deal with cell ranges in a spreadsheet in an efficient manner.
- Pure also provides a bridge to Octave so that you can call arbitrary Octave functions using this extension.
- Gnumeric/Pure offers support for rendering OpenGL scenes in Gnumeric frame widgets, via Pure's own OpenGL interface.

- Pure also has built-in support for lazy data structures and thus allows you to handle potentially infinite amounts of data such as the list of all prime numbers. Gnumeric/Pure lets you turn such lazy values into asynchronous data sources computed in the background, which update the spreadsheet automatically as results become available.
- Last but not least, Pure is compiled to native code on the fly. This means that, while startup times are a bit longer due to Pure's JIT compiler kicking in (you'll notice this if you open a spreadsheet with Pure functions), the resulting compiled code then typically executes *much* faster than equivalent interpreted Perl and Python code.

Once the plugin loader is installed and enabled, you can try the Pure functions in the provided examples and start adding your own plugin scripts. As of version 0.12, there's a new helper script `pure-gnm` which generates the required plugin.xml files to make this easy. Various examples can be found in the examples folder in the distribution, which should help you to get started with Gnumeric/Pure fairly quickly.

For more advanced uses, Gnumeric/Pure also provides a programming interface which lets you do various special tasks such as modifying entire ranges of cells with one Pure call, calling Gnumeric functions from Pure, and setting up asynchronous data sources and OpenGL frames. The manual explains all this in detail.

Note: This manual assumes that you're already familiar with Gnumeric as well as the Pure language and its programming environment. If not then you should consult the corresponding documentation to learn more about these.

12.2 Copying

Copyright (c) 2009-2012 by Albert Graef.

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

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

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

12.3 Installation

Get the latest source from http://pure-lang.googlecode.com/files/gnumeric-pure-0.12.tar.gz.

Obviously, you need to have both Pure and Gnumeric installed. Pure 0.36 and Gnumeric 1.9.13 or later are known to work. We recommend Gnumeric 1.9.14 or later since it has improved support for GUI widgets. (Older Gnumeric versions probably work as well if you're willing to fiddle with the Makefile and/or the sources. See the beginning of the Makefile for related information.)

Note: At present, Gnumeric/Pure will *not* work with the current Gnumeric 1.11 development series which is based on GTK3 and reportedly has a different plugin interface. (At least nobody has tried porting Gnumeric/Pure to it yet.) So you'll have to stick to the stable Gnumeric series for the time being.

As shipped, the Makefile is set up to build Gnumeric/Pure with OpenGL support, which requires that you have the OpenGL libraries as well as GtkGLExt (the Gtk OpenGL extension) installed. These should be readily available on most systems, but you can also disable this feature by invoking `make` as `make GLDEPS=`.

Run `make` to compile the software. You might have to adjust the settings at the beginning of the Makefile to make this work. Once the compile goes through, you should now have a `pure_loader.so` file in the `pure-loader` subdirectory. You can install the plugin and related stuff with `sudo make install` in the global Gnumeric plugin directory, or if you prefer to install it into your personal plugin directory then run `make install-local` instead. (The latter is recommended if you plan to customize any of the sample plugin scripts included in the distribution for your purposes.)

Typically, `make install` and `make install-local` will install the plugins into the following directories by default (here and in the following `<version>` denotes the version of Gnumeric you have installed):

- System-wide installations go into `/usr/local/lib/gnumeric/<version>/plugins` or similar, depending on Gnumeric's installation prefix (usually either `/usr/local` or `/usr`).
- User-specific installations go into `~/.gnumeric/<version>/plugins`.

The Makefile tries to guess the installation path and version number of Gnumeric on its own. If it guesses wrong, you can change these using the Makefile variables `prefix` and `gnmversion`, respectively. For instance:

```
$ make prefix=/usr gnmversion=1.10.13
```

In either case, `make install` also installs the `pure-gnm` helper script under the Pure installation prefix. (This is a little convenience script to generate the `plugin.xml` files used by Gnumeric to load a plugin; see Defining Your Own Functions for details.)

If `make install` doesn't work for some reason, you can also just copy the `pure-func`, `pure-glfunc` and `pure-loader` directories manually to your Gnumeric plugin directory. You can still run `make install` in the `pure-gnm` subdirectory to get the `pure-gnm` script installed in this case.

12.4 Setup

Once Gnumeric/Pure has been properly installed, you should see it in Gnumeric's Tools/Plug-ins dialog. There are actually two main entries, one labelled "Pure functions" and the other one labelled "Pure plugin loader". You need to enable both before you can start using Pure functions in your Gnumeric spreadsheets. There's also a third entry labelled "Pure OpenGL functions" which you might want to enable if you want to try the OpenGL capabilities (this will only work if you built Gnumeric/Pure with OpenGL support and have Pure's OpenGL module installed; see OpenGL Interface for details).

Gnumeric doesn't provide much in the way of GUI customization options right now, but at least it's possible for plugins to install and configure additional menu and toolbar options. Gnumeric/Pure adds three additional options to the Tools menu which allow you to stop asynchronous data sources, reload Pure scripts and edit them. After installation, the definitions of these items can be found in the `pure-loader/pure-ui.xml` file in your Gnumeric plugin directory. Have a look at this file and edit is as desired. E.g., if you want to put the Pure-related options into a submenu and enable toolbar buttons for these options, then your `pure-ui.xml` file should look as follows:

```
<ui>
  <menubar>
    <menu name="Tools" action="MenuTools">
      <separator/>
      <menu name="Pure" action="PureMenu">
        <menuitem action="PureStop"/>
        <menuitem action="PureReload"/>
        <menuitem action="PureEdit"/>
      </menu>
    </menu>
  </menubar>
  <toolbar name="StandardToolbar">
    <separator/>
    <toolitem action="PureStop"/>
    <toolitem action="PureReload"/>
    <toolitem action="PureEdit"/>
  </toolbar>
</ui>
```

12.5 Basic Usage

With Pure/Gnumeric installed and enabled, you should be ready to join the fun now. Start up Gnumeric, click on a cell and invoke the "f(x)" dialog. The Pure functions available for use are shown in the "Pure" category. E.g., click on `pure_hello`. Now the Pure interpreter will be loaded and the function description displayed. Click "Ok" to select the `pure_hello` function and then "Ok" again to actually insert the function call (without arguments) into the current cell. You should now be able to read the friendly greeting returned by the function.

Of course, you can also enter the function call directly as a formula into a cell as usual. Click

on a cell, then enter the following:

```
=pure_hello(getenv("USER"))
```

The greeting should now be displayed with your login name in it.

Play around a bit with the other Pure functions. These functions are nothing special; they are just ordinary Pure functions which are defined by the `pure_func.pure` script in the `pure-func` subdirectory of your Gnumeric plugin directory. You can have a look at them by invoking the "Edit Pure Scripts" option which gets added to the Tools/Pure menu once the Pure plugin loader is enabled. (This will invoke the emacs editor by default, or the editor named by the `EDITOR` environment variable. You can set this environment variable in your shell's startup files.) The Tools/Pure menu contains a second Pure-related option, "Reload Pure Scripts" which can be used to quickly reload all loaded Pure scripts after edits; more about that later.

Please note that most of the functions in `pure_func.pure` are rather useless, they are only provided for illustrative purposes. However, there are some useful examples in there, too, in particular:

- `pure_eval` lets you evaluate any Pure expression, given as a string in its first argument. E.g., try something like `=pure_eval("foldl (+) 0 (1..100)")`. Additional parameters are accessible as `x!0`, `x!1`, etc. For instance: `=pure_eval("x!0+x!1",A1,B1)`.
- `pure_echo` just displays its arguments as a string in Pure syntax, as the interpreter sees them. This is useful for debugging purposes. E.g., `=pure_echo(A1:B10)` shows the given range as a Pure matrix.
- `pure_shell` is a variation of `pure_eval` which executes arbitrary Pure code and returns the last evaluated expression (if any) as a string. This is mainly provided as a convenience to create an "interactive Pure shell" which lets you evaluate Pure code inside Gnumeric. To these ends, simply prepare a text cell for entering the code to be evaluated, and then apply `pure_shell` on this text cell in another cell to display the result.

A spreadsheet showing most of the predefined functions in action can be found in `pure-examples.gnumeric` example distributed with Gnumeric/Pure.

12.6 Interactive Pure Shell

The `pure-examples.gnumeric` spreadsheet also includes an instance of `pure_shell` which lets you evaluate arbitrary Pure code in the same interpreter instance that executes Gnumeric/Pure functions. This is very helpful if you're developing new Pure functions to be used in Gnumeric. It also lets you use Gnumeric as a kind of GUI frontend to the Pure interpreter. You can try this now. Open the `pure-examples` spreadsheet in Gnumeric and enter the following into the input cell of the Pure shell:

```
> scanl (+) 0 (1..20)
  [0,1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210]
```

Note that here and in the following the prompt > indicates a Pure expression to be evaluated in *Gnumeric* (rather than the standalone Pure interpreter), which is followed by another line indicating the result (printed in the output cell below the input cell of the Pure shell). You can find the Pure shell at the bottom of the first sheet in `pure-examples`, see the screenshot below. For your convenience, there's also a second, bigger one on the second sheet. You might want to copy this over to a separate spreadsheet which you can use as a scratchpad for experimentation purposes.

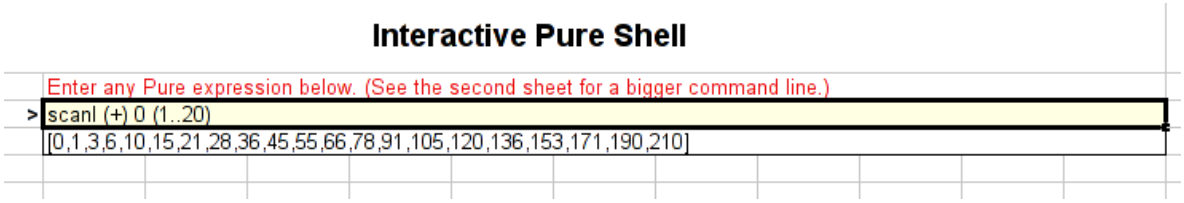

Figure 12.1: The Pure shell.

Also note that this is in fact *Pure code* (not a Gnumeric formula) being evaluated there. You can execute any Pure code, including Pure declarations, so you can type:

```
> using system; puts "Hello, world!";
  14
```

This prints the string `"Hello, world!"` on standard output, visible in the terminal window where you launched Gnumeric. Here is another example, showing how you can invoke any function from the C library, by declaring it as a Pure `extern` function:

```
> extern int rand(); [rand | i = 1..5];
  [1810821799,2106746672,1436605662,1363610028,695042099]
```

All functions in the Pure prelude are readily available in the Gnumeric Pure shell, as well as the functions defined in `pure_func.pure` and its imports, including the programming interface described in Advanced Features. For instance, here's how you can retrieve a cell value from the current sheet:

```
> get_cell "A1"
  "Gnumeric/Pure Examples"
```

Using `call` (see Calling Gnumeric from Pure), you can also invoke any Gnumeric function:

```
> call "product" (1..10)
  3628800.0
```

12.7 Defining Your Own Functions

After playing around with `pure_func.pure` and the interactive Pure shell for a while, of course you will want to write your own functions, that's what this extension is about after all! This section shows you how to do this.

12.7.1 Creating a Simple Plugin

Let's consider a simple example: the factorial function. In Pure this function can be implemented as follows:

```
factorial [x] = foldl (*) 1 (1..x);
```

Note the list bracket around the argument `x`. You wouldn't normally pass a single numeric argument that way in Pure, but this is needed here since by default Gnumeric passes arguments as a list to a Pure function. There are other ways to configure the call interface to Pure functions, but these require that we tell Gnumeric about the number and types of arguments, see Gnumeric/Pure Interface below. For the moment let's stick to the default scheme, however, in order to keep things simple.

Put the above definition into a script file, say, `myplugin.pure`. Next we need to create a `plugin.xml` file to tell Gnumeric about our plugin and which functions it provides. While these files can be written by hand, this is tedious and error-prone. Fortunately, recent Gnumeric/Pure versions provide the `pure-gnm` helper script which makes this quite easy. To use `pure-gnm` with our plugin script, we have to add a special "hashbang" comment block to our script which supplies the needed information. In our case, this might look as follows:

```
#! N: My Pure functions
#! C: Pure
#! D: My Pure functions.
#! F: factorial
```

You can add this comment block anywhere in your plugin script, but usually it is placed near the beginning. The different fields have the following meaning:

- `N`: the name of the plugin
- `C`: the function category
- `D`: a more detailed description of the plugin
- `F`: a whitespace-delimited list of Pure function names

The contents of the `N` and `D` fields (name and description) are visible in Gnumeric's "Plugin Manager" dialog. You should specify at least the name field (otherwise the plugin will be displayed as "Unnamed" in the dialog), while the description is optional (if you don't specify one, the description of the plugin will be empty). The `C` field denotes the category under which the functions listed in the `F` field will be shown in Gnumeric's "f(x)" dialog; if you don't specify this, the functions will be in the "Unknown" category. The `F` field is the most crucial part. It must contain all Pure functions defined in the plugin script or its imports that you want to be visible in Gnumeric, so you have to keep this in sync with the actual function definitions in the script; if you don't specify this, the plugin will provide no functions at all.

The `D` and `F` fields can also be split into multiple lines (each prefixed with the "hashbang" comment marker and the corresponding field identifier) if necessary.

So our `myplugin.pure` script now looks like this:

```
#! N: My Pure functions
#! C: Pure
#! D: My Pure functions.
#! F: factorial

factorial [x] = foldl (*) 1 (1..x);
```

Once you've added the comment block, you can generate the `plugin.xml` file for the plugin simply as follows:

```
$ pure-gnm myplugin.pure > plugin.xml
```

Note that by default `pure-gnm` writes the `plugin.xml` file to standard output which is useful if you want to check the generated file first. To actually create the file, we simply redirect the output to `plugin.xml`.

You'll have to redo this every time your plugin changes (i.e., you've added new functions or deleted or renamed old ones, or changed the name or description of the plugin). It's easy to automate this step using `make`. E.g., the following Makefile will do the trick:

```
myplugin = myplugin.pure

all: plugin.xml

plugin.xml: $(myplugin)
        pure-gnm $< > $@

clean:
        rm -f plugin.xml
```

Now you can just run `make` in the plugin directory and it will rebuild the `plugin.xml` file as needed.

12.7.2 The plugin.xml File

The `plugin.xml` file resulting from the previous step looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="Gnumeric_myplugin">
  <information>
    <name>My Pure functions</name>
    <description>My Pure functions.</description>
    <require_explicit_enabling/>
  </information>
  <loader type="Gnumeric_PureLoader:pure">
    <attribute name="module_name" value="myplugin.pure"/>
  </loader>
  <services>
    <service type="function_group" id="myplugin">
      <category>Pure</category>
      <functions>
```

```
        <function name="factorial"/>
      </functions>
    </service>
  </services>
</plugin>
```

You can also edit this file by hand if you know what you're doing; in that case, please check the Gnumeric documentation for details about the format of these files. The template used by `pure-gnm` to generate these files can be found in the source distribution (see `plugin.xml` in the `pure-gnm` folder) or under `/usr/local/lib/pure-gnm` after installation. You can edit this file (carefully!) in order to implement global changes that you want to be in every `plugin.xml` file generated by `pure-gnm`.

Two specific items that you might want to edit by hand are the `<require_explicit_enabling/>` tag and the `id` properties of the `<plugin>` and `<service>` tags:

- The `<require_explicit_enabling/>` tag indicates that Gnumeric shouldn't enable the new plugin until you explicitly tell it to. You can remove that line if you want Gnumeric to automatically enable new plugins as they are added to the system.
- The `pure-gnm` script automatically derives the `id` properties of the `<plugin>` and `<service>` tags from the name of the plugin script, which is a sensible default in most cases. However, you might have to change these identifiers if they happen to collide with other Gnumeric plugins and services. This can be done by either editing the generated `plugin.xml` file or by renaming the plugin script accordingly.

Note that the only really Pure-specific part in the xml file is the loader description which also names the Pure script implementing the plugin in the value of the `module_name` attribute. In this case this is just `"myplugin.pure"`. This path is taken relative to the directory containing the `plugin.xml` file, but you can also specify an absolute path there if you want to keep the plugin script elsewhere. To achieve this with `pure-gnm`, you can just invoke it with an absolute path name, e.g.:

```
$ pure-gnm $PWD/myplugin.pure > plugin.xml
```

Now you can move the `plugin.xml` file whereever you like and still have Gnumeric find the script file in its prescribed location. Again, this can be automatized using `make` fairly easily; we'll return to that in the following section.

12.7.3 Loading the Plugin

We now have the plugin script `myplugin.pure` and the `plugin.xml` file in the same directory, say, `/some/path/myplugin`. We still need to tell Gnumeric about the new plugin, though, so that it can find it. Unfortunately, the mechanics of making a plugin known to Gnumeric are somewhat involved, so we discuss the necessary steps in detail below. There are basically three ways you can go about this:

- If you want to keep plugin script and the `plugin.xml` file where they are, you'll have to change Gnumeric's plugin path so that it includes the *parent* directory `/some/path` (not `/some/path/myplugin`). This is done by adding the directory under the Directories tab in Gnumeric's Tools/Plug-ins dialog, after which you'll have to restart Gnumeric so that it picks up the changes in the plugin search path.
- Second, you can also move or copy the entire `/some/path/myplugin` directory to your personal Gnumeric plugin folder (usually `~/.gnumeric/<version>/plugins`). Gnumeric will always search this directory for new plugins by default, so modifying the plugin search path is not necessary. However, keeping the plugin script in a hidden location in your home directory may not be very convenient if you want to modify the script later.
- Third, you can get the best of both previous methods by keeping the plugin script where it is and copying just the `plugin.xml` file to your personal Gnumeric plugin folder.

The third method tends to be the easiest, but note that it requires that the plugin script needs to be specified as an absolute path (as sketched out previously). Fortunately, it's fairly easy to automate this with `make`. The following requires GNU make to work, and you'll also need to have the Gnumeric development files installed, so that the Gnumeric version can be determined easily with a shell command. These rules are to be added to the end of the Makefile described previously under Creating a Simple Plugin.

```
gnmversion=$(shell pkg-config --modversion libspreadsheet-1.10)
plugindir=$(HOME)/.gnumeric/$(gnmversion)/plugins/myplugin

install: $(myplugin)
        test -d $(plugindir) || mkdir -p $(plugindir)
        pure-gnm $(CURDIR)/$< > $(plugindir)/plugin.xml

uninstall:
        rm -rf $(plugindir)
```

Now you can just run `make install` to make the plugin known to Gnumeric. Note that we also added a rule that allows you to uninstall the plugin if it isn't needed any more.

In any case, once you fire up Gnumeric again, the new plugin should be listed as "My Pure functions" on the Plugin List tab in the Tools/Plug-ins dialog. Check it to enable it. The `factorial` function defined in the plugin should now be available and ready to be called just like any other Gnumeric function. For instance, type this into a cell to have the factorial of 10 computed:

```
=factorial(10)
```

Also try saving the spreadsheet and loading it again after restarting Gnumeric. The plugin will now be loaded automatically and the spreadsheet should display the proper value of the factorial.

Note: Once you start playing around with your own Pure plugins, you may run into one common mishap: You open an existing spreadsheet without having enabled the plugins it

uses. An easily visible symptom of that is that you'll see cells showing the `#NAME?` error. You will then have to enable those plugins again *and* reload the spreadsheet afterwards, so that everything is recalculated properly.

In contrast, just changing the body of a function in a plugin usually needs neither a restart of Gnumeric nor a reloading of the spreadsheet. In this case it's often sufficient to reload all scripts with the "Reload Pure Scripts" option in the Tools/Pure menu, after which you can use "Recalculate" (F9) to recompute the spreadsheet.

It is also worth mentioning here that the Pure loader can load multiple Pure plugins (and of course each plugin can provide as many functions as you want). You only need to tell Gnumeric about them after creating the scripts and `plugin.xml` files and placing them into corresponding plugin directories. Just enable the ones that you want in Tools/Plug-ins. All scripts are loaded in the same Pure interpreter (and thus are treated like one big script) so that functions in one script can use the function and variable definitions in another. If you need to access the definitions in the `pure_func.pure` "mother script", you can also just import it into your scripts with a `using` clause, i.e.: `using pure_func;`

Another important point is that a Pure plugin script is always loaded in the directory where it is located, as indicated by the corresponding `plugin.xml` file, even if it is different from the plugin directory. That is, the current working directory (which is normally the directory that Gnumeric was started in) is temporarily set to the directory holding the plugin script while the script is being loaded. This enables the script to find imported scripts and other files (such as media files or scripts written in other languages) that it may need at load time. This wasn't needed in this simple example, but you can find other examples in the Gnumeric/Pure distribution which make good use of this feature.

12.7.4 Spicing It Up

Our plugin example is now essentially complete, but in order to make it really convenient to use, we may want to add some information about how the `factorial` function is to be called in Gnumeric. Gnumeric doesn't keep this kind of information in the `plugin.xml` file, but expects it to be provided by the plugin itself. In the Gnumeric/Pure interface this can be done by adding a rule for the `gnm_info` function. In our example we tell Gnumeric that `factorial` expects a single numeric argument. While we're at it, we might as well add some helpful documentation to be displayed in Gnumeric's "f(x)" dialog. The details of this are described in the following section, but to give you a sneak preview, here's a beefed-up version of our script which implements all this (you can also find this version of the example along with a GNU Makefile in the Gnumeric/Pure distribution):

```
#! N: My Pure functions
#! C: Pure
#! D: My Pure functions.
#! F: factorial

factorial x = foldl (*) 1 (1..x);

using pure_func; // for the gnm_help function
```

```
gnm_info "factorial" = "f", gnm_help "factorial:factorial of a number"
 ["x:number"] "Computes the factorial of @{x}." [] ["=factorial(10)"] [];
```

Fire up Gnumeric again, press the "f(x)" button and select `factorial` under the `Pure` category. The "f(x)" dialog should now display the additional information we added above. Also note that Gnumeric now knows that this function is supposed to be called with exactly one `f` (numeric) argument. Therefore the list brackets around the argument of `factorial` aren't needed any more, so don't forget to remove them, as shown in the above code sample.

This completes our little example. As an exercise, you're invited to add more functions on your own. (Don't forget to change the `#! F` line accordingly and rerun `pure-gnm` when you do this, so that Gnumeric knows about the new functions.) It also pays off to take a look at some of the other included examples, you can find these in the `examples` folder of the distribution tarball.

12.8 Gnumeric/Pure Interface

We already explained in the previous section that, when a Pure function is called from Gnumeric, it receives its arguments in a list by default. However, it is possible to tell Gnumeric about the expected arguments of the function and also specify a help text to be displayed in the "f(x)" dialog, by giving a definition of the `gnm_info` function as explained below.

Note that `gnm_info` is really an ordinary Pure function. Thus, rather than hardcoding this information as static text (such as the "docstrings" used in Gnumeric's Python extension), the function descriptions can also be constructed dynamically in corresponding Pure code. This offers an opportunity for programmatic customizations. But note that the `gnm_info` function will only be invoked when the plugin script is loaded, so once that is done the function description remains the same for the entire Gnumeric session.

12.8.1 Function Descriptions

To describe a given function to Gnumeric, define `gnm_info "<name>"` (where `<name>` is the name of the function) as a pair with the following elements:

- The first element, a string, gives the signature of the function. E.g., `""` denotes a function without arguments, `"f"` a function taking a single float parameter, `"fs"` a function taking a float and a string argument (in that order), etc. Optional parameters can be indicated using `|`, as in `"ff|s"` (two non-optional floats, followed by an optional string). See below for a complete list of the supported parameter types.
- The second element is a list of hash pairs `key=>text` which together make up the help text shown in Gnumeric's "f(x)" dialog. You should at least specify the function name along with a short synopsis here, e.g. `GNM_FUNC_HELP_NAME => "frob:the frob function"`. Parameter descriptions take the form `GNM_FUNC_HELP_ARG => "x:integer"`. There are a number of other useful elements, see below for details.

Both the signature and the function description are optional. That is, `gnm_info` may return either just a signature string, or a list of hash pairs with the function description, or both. The signature defaults to a variadic function which takes any number of parameters of any type (see below), and the description defaults to some boilerplate text which says that the function hasn't been documented yet.

Note that if no signature is given, then the function accepts any number of parameters of any type. In that case, or if there are optional parameters, the function becomes variadic and the (optional) parameters are passed as a Pure list (in addition to the non-optional parameters).

Here's the list of valid parameter types, as they are documented in the Gnumeric sources:

```
f : float           no errors, string conversion attempted
b : boolean         identical to f
s : string          no errors
S : scalar          any non-error scalar
E : scalar          any scalar, including errors

r : cell range      content may not be evaluated yet
A : area            array, range (as above), or scalar
? : anything        any value (scalars, non-scalars, errors, whatever)
```

The keys used in the function description may be any of the following, along with sample text for each type of field:

```
GNM_FUNC_HELP_NAME         => "name:synopsis"
GNM_FUNC_HELP_ARG          => "name:parameter description"
GNM_FUNC_HELP_DESCRIPTION  => "Long description."
GNM_FUNC_HELP_NOTE         => "Note."
GNM_FUNC_HELP_EXAMPLES     => "=sample_formula()"
GNM_FUNC_HELP_SEEALSO      => "foo,bar,..."
```

The following keys are only supported in the latest Gnumeric versions:

```
GNM_FUNC_HELP_EXTREF       => "wiki:en:Trigonometric_functions"
GNM_FUNC_HELP_EXCEL        => "Excel compatibility information."
GNM_FUNC_HELP_ODF          => "OpenOffice compatibility information."
```

Note that inside the descriptions, the notation `@{arg}` (`@arg` in older Gnumeric versions) can be used to refer to a parameter value. For instance, here's a sample description for a binary function which also includes a help text:

```
gnm_info "pure_max" = "ff",
[GNM_FUNC_HELP_NAME     => "pure_max:maximum of two numbers",
 GNM_FUNC_HELP_ARG      => "x:number",
 GNM_FUNC_HELP_ARG      => "y:number",
 GNM_FUNC_HELP_DESCRIPTION =>
 "Computes the maximum of two numbers @{x} and @{y}.",
 GNM_FUNC_HELP_EXAMPLES => "=pure_max(17,22)"];
```

As you can see, the function descriptions are a bit unwieldy, so it's convenient to construct them using this little helper function defined in `pure_func.pure`:

```
gnm_help name::string args descr::string notes examples see_also =
  [GNM_FUNC_HELP_NAME         => name] +
  [GNM_FUNC_HELP_ARG          => x | x::string = args ] +
  [GNM_FUNC_HELP_DESCRIPTION  => descr ] +
  [GNM_FUNC_HELP_NOTE         => x | x::string = notes ] +
  [GNM_FUNC_HELP_EXAMPLES     => x | x::string = examples ] +
  (if null see_also then [] else
   [GNM_FUNC_HELP_SEEALSO     => join "," see_also]);
```

Now the description can be written simply as follows:

```
gnm_info "pure_max" = "ff", gnm_help "pure_max:maximum of two numbers"
  ["x:number", "y:number"]
  "Computes the maximum of two numbers @{x} and @{y}."
  [] ["=pure_max(17,22)"] [];
```

Since this function only has fixed arguments, it will be called in curried form, i.e., as `pure_max x y`. For instance, the actual definition of `pure_max` may look as follows:

```
pure_max x y = max x y;
```

Conversely, if no signature is given, then the function accepts any number of parameters of any type, which are passed as a list. For instance:

```
gnm_info "pure_sum" = gnm_help "pure_sum:sum of a collection of numbers"
  [] "Computes the sum of a collection of numbers."
  [] ["=pure_sum(1,2,3,4,5,6)"] ["pure_sums"];
```

Here the function will be called as `pure_sum [x1,x2,...]`, where `x1`, `x2`, etc. are the arguments the function is invoked with. Note that in this case there may be any number of arguments (including zero) of any type, so your function definition must be prepared to handle this. If a function does not have a `gnm_info` description at all then it is treated in the same fashion. The `pure_func.pure` script contains some examples showing how to write functions which can deal with any numbers of scalars, arrays or ranges, see the `pure_sum` and `pure_sums` examples. These employ the following `ranges` function to "flatten" a parameter list to a list holding all denoted values:

```
ranges xs = cat [ case x of _::matrix = list x; _ = [x] end | x = xs ];
```

E.g., the `pure_sum` function can now be defined as follows:

```
pure_sum xs = foldl (+) 0 (ranges xs);
```

A function may also have both fixed and optional arguments (note that in what follows we're going to omit the detailed function descriptions for brevity):

```
gnm_info "foo" = "ff|ff";
```

In this case the fixed arguments are passed in curried form as usual, while the optional parameters are passed as a list. That is, `foo` may be called as `foo x y []`, `foo x y [z]` or `foo x y [z,t]`, depending on whether it is invoked with two, three or four arguments.

12.8.2 Conversions Between Pure and Gnumeric Values

The marshalling of types between Gnumeric and Pure is pretty straightforward; basically, Pure numbers, strings and matrices map to Gnumeric numbers, strings and arrays, respectively. The following table summarizes the available conversions:

Pure	Gnumeric
`gnm_error "#N/A"`	error
`4711`, `4711L`, `4711.0`	scalar (number)
`"Hello world"`	string
`()`	empty
`(1,2,3)`	array
`[1,2,3]`	array
`{1,2,3;4,5,6}`	array (or cell range)
`"A1:B10"`	cell range (`"r"` conversion)

These conversions mostly work both ways. Note that on input, cell ranges are usually passed as matrices to Pure functions (i.e., they are passed "by value"), unless the function signature specifies a `"r"` conversion in which case the cell ranges themselves are passed to the function in string form. (Such values can also be passed on to Gnumeric functions which expect a cell range (`"r"`) parameter, see Calling Gnumeric from Pure below.)

Conversely, matrices, lists and tuples all become Gnumeric arrays on output, so usually you'll want to enter these as array functions (`Ctrl-Shift-Enter` in Gnumeric). As a special case, the empty tuple can be used to denote empty cell values (but note that empty Gnumeric values may become zeros when passed as float or array arguments to Pure functions).

Another special case is a term of the form `gnm_error msg`, where `msg` is a string value indicating a Gnumeric error value such as `"#N/A"`, `"#NAME?"`, `"#NULL!"`, etc. When returned by a plugin function, the error text will be displayed in the corresponding Gnumeric cell.

If a Pure function returns a value that doesn't match any of the above then it is converted to a string in Pure expression syntax and that string is returned as the result of the function invocation in Gnumeric. This makes it possible to return any kind of symbolic Pure value, but note that if such a value is then fed into another Pure function, that function will have to convert the string value back to the internal representation if needed; this can be done very conveniently using Pure's `eval` function, see the Pure documentation for details.

12.9 Advanced Features

This section explains various additional features provided by the Gnumeric/Pure interface that should be useful for writing your own functions. Note that for your convenience all functions discussed in this section are declared in `pure_func.pure`.

12.9.1 Calling Gnumeric from Pure

It is possible to call Gnumeric functions from Pure using the `call` function which takes the name of the function (a string) as its first, and the parameters as the second (list) argument. For instance:

```
gnm_info "gnm_bitand" = "ff";
gnm_bitand x y = call "bitand" [x,y];
```

Note that `call` is an external C function provided by Gnumeric/Pure. If you want to use it, it must be declared in your Pure script as follows:

```
extern expr* pure_gnmcall(char* name, expr* args) = call;
```

However, `pure_func.pure` already contains the above declaration, so you don't have to do this yourself if you import `pure_func.pure` in your scripts.

Also note that `call` doesn't do any of Gnumeric's automatic conversions on the parameters, so you have to pass the proper types of arguments as required by the function.

12.9.2 Accessing Spreadsheet Cells

Gnumeric/Pure provides the following functions to retrieve and modify the contents of spreadsheet cells and ranges of such cells:

```
extern expr* pure_get_cell(char* s) = get_cell;
extern expr* pure_get_cell_text(char* s) = get_cell_text;
extern expr* pure_get_cell_format(char* s) = get_cell_format;
extern expr* pure_set_cell(char* s, expr *x) = set_cell;
extern expr* pure_set_cell_text(char* s, expr *x) = set_cell_text;
extern expr* pure_set_cell_format(char* s, expr *x) = set_cell_format;
extern expr* pure_get_range(char* s) = get_range;
extern expr* pure_get_range_text(char* s) = get_range_text;
extern expr* pure_get_range_format(char* s) = get_range_format;
extern expr* pure_set_range(char* s, expr *x) = set_range;
extern expr* pure_set_range_text(char* s, expr *x) = set_range_text;
extern expr* pure_set_range_format(char* s, expr *x) = set_range_format;
```

For instance, here's how you use these functions to write and then read some cell values (try this in the interactive Pure shell):

```
> set_cell "A14" 42
  ()
> get_cell "A14"
  42.0
> set_range "A14:G14" $ scanl (*) 1 (1..6)
  ()
> get_range "A14:G14"
  {1.0,1.0,2.0,6.0,24.0,120.0,720.0}
> set_cell_text "A14" "=sum(B14:G14)"
  ()
```

```
> get_cell "A14"
  873.0
> get_cell_text "A14"
  "=sum(B14:G14)"
> get_range_text "A14:G14"
  {"=sum(B14:G14)","1","2","6","24","120","720"}
```

Note that while the `set_cell` function sets the given cell to a constant value, `set_cell_text` also allows you to store a formula in a cell which will then be evaluated as usual. Similarly, `get_cell` retrieves the cell value, while `get_cell_text` yields the text in the cell, as entered by the user (which will either be a formula or the textual representation of a constant value). The `set_range`, `set_range_text`, `get_range` and `get_range_text` functions work analogously, but are used to manipulate entire ranges of cells, which can be set from Pure tuples, lists or matrices, and retrieved as Pure matrices.

Functions to retrieve and change the cell format are also provided (watch the contents of the cell `A14` change its color to blue on entering the first expression):

```
> set_cell_format "A14" "[Blue]0.00"
  ()
> get_range_format "A14:C14"
  {"[Blue]0.00","General","General"}
```

There are also functions to get the position of the "current" cell (i.e., the cell from which a Pure function was called), and to translate between cell ranges in Gnumeric syntax and the corresponding internal representation consisting of a pointer to a Gnumeric sheet and the cell or range coordinates:

```
extern expr* pure_this_cell() = this_cell;
extern expr* pure_parse_range(char* s) = parse_range;
extern expr* pure_make_range(expr* x) = make_range;
```

Examples:

```
> this_cell
  "B4"
> parse_range this_cell
  #<pointer 0x875220>,1,3
> make_range (NULL,0,0,10,10)
  "Sheet2!A1:K11"
```

12.9.3 Asynchronous Data Sources

Gnumeric/Pure makes it easy to set up asynchronous data sources which draw values from a Pure computation executed in a background process. This facility is useful to carry out lengthy computations in the background while you can continue to work with your spreadsheet. It also allows you to process incoming data and asynchronous events from special devices (MIDI, sensors, stock tickers, etc.) in (soft) realtime.

To do this, you simply pass an expression to the `datasource` function. This is another external C function provided by Gnumeric/Pure, which is declared in `pure_func.pure` as follows:

```
extern expr* pure_datasource(expr* x) = datasource;
```

The argument to `datasource` is typically a thunk or stream (lazy list) which is to be evaluated in the background. The call to `datasource` initially returns a `#N/A` value (`gnm_error "#N/A"`) while the computation is still in progress. The cell containing the data source then gets updated automatically as soon as the value becomes available, at which point the `datasource` call now returns the computed value. E.g., here's how you would wrap up a lengthy calculation as a thunk and submit it to `datasource` which carries out the computation as a background task:

```
gnm_info "pure_frob" = "f";
pure_frob x = datasource (lengthy_calculation x&);
lengthy_calculation x = sleep 3 $$ foldl (*) 1 (1..x);
```

Note that a cell value may draw values from as many independent data sources as you want, so the definition of a cell may also involve multiple invocations of `datasource`:

```
gnm_info "pure_frob2" = "ff";
pure_frob2 x y = datasource (lengthy_calculation x&),
  datasource (lengthy_calculation y&);
```

Special treatment is given to (lazy) lists, in this case `datasource` returns a new value each time a list element becomes available. For instance, the following function uses an infinite stream to count off the seconds starting from a given initial value:

```
gnm_info "pure_counter" = "f";
pure_counter x = datasource [sleep (i>x) $$ i | i = x..inf];
```

You can also try this interactively in the Pure shell:

```
> datasource [sleep (i>0) $$ i | i = 0..inf]
  0
  1
  ...
```

Here's another example for the Pure shell which prints the prime numbers:

```
> datasource primes with primes = sieve (2..inf);
    sieve (p:qs) = p : (sleep 1 $$ sieve [q | q = qs; q mod p])& end
  2
  3
  5
  ...
```

Note that when processing a lazy list, the cell containing the call will keep changing as long as new values are produced (i.e., forever in this example). The "Stop Data Sources" option in the Tools/Pure menu can be used to stop all active data sources. "Reload Pure Scripts"

also does this. You can then restart the data sources at any time by using "Recalculate" (F9) to recompute the spreadsheet.

Also note that because of the special way that datasource handles list values, you cannot return a list directly as the result of datasource, if it is to be treated as a single result. Instead, you'll have to wrap the result in a singleton list (e.g., datasource [[lengthy_calculation x,lengthy_calculation y]&]), or return another aggregate (i.e., a matrix or a tuple).

Finally, note that when the arguments of a call involving datasource change (because they depend on other cells which may have been updated), the computation is automatically restarted with the new parameters. The default behaviour in this case is that the entire computation will be redone from scratch, but it's also possible to wrap up calls to datasource in a manner which enables more elaborate communication between Gnumeric and background tasks initiated with datasource. This is beyond the scope of this manual, however, so we leave this as an exercise to the interested reader.

12.9.4 Triggers

In addition to asynchronous data sources, the trigger function is provided to compute values or take actions depending on some external condition, such as the availability of data on a special device or the creation of some widget (see the next section):

```
extern expr* pure_trigger(int timeout, expr* cond, expr *val, expr *data)
  = trigger;
```

Thus a typical invocation of the function looks as follows:

```
trigger timeout condition value data
```

The condition and value arguments are callback functions which get invoked by trigger, passing them the given data argument. The trigger reevaluates the given condition in regular intervals (1 second in the current implementation) and, as soon as it becomes true, computes the given value and returns that value as the result of the trigger call. As long as the condition doesn't hold, trigger returns a #N/A value (gnm_error "#N/A"). Note that, in difference to datasource, both the condition and the value are computed in Gnumeric (rather than a child process), so that it is possible to access the current information in the loaded spreadsheet.

The timeout value determines how often the condition is checked. If it is positive, the condition will be reevaluated timeout+1 times (once initially, and then once per second for a total duration of timeout seconds). If it is negative, the trigger never times out and the condition will be checked repeatedly until the trigger expression is removed (or Gnumeric is exited). In either case value data will be recomputed each time condition data yields true. (This is most useful if the computed value, as a side effect, arranges for the condition to become false again afterwards.) Finally, if timeout is zero then the trigger fires at most once, as soon as the condition becomes true, at which point value data is computed just once.

Here's a (rather useless) example of a trigger which fires exactly once, as soon as a certain cell goes to a certain value, and then modifies another cell value accordingly:

```
> trigger 0 (\_->get_cell "A14"=="Hello") (\_->set_cell "A15" "World") ()
```

Now, as soon as you type `Hello` in the cell A14, the trigger will print `World` in cell A15. Note that the `data` argument isn't used here. A more useful example will be discussed in the following section.

12.9.5 Sheet Objects

Gnumeric offers some kinds of special objects which can be placed on a sheet. This comprises the chart and image objects which can be found in the "Insert" menu, as well as a number of useful graphical elements and GUI widgets on the "Object" toolbar, accessible via "View/Toolbars". The latter are also useful for providing control input to Pure functions.

Gnumeric/Pure provides the following function to retrieve information about the special objects in a spreadsheet:

```
extern expr* pure_sheet_objects() = sheet_objects;
```

For instance, with one button object in your spreadsheet, the output of `sheet_objects` might look like this:

```
> sheet_objects
  [("Sheet1","button","Push Me!","A11",[#<pointer 0x2a1dcd0>])]
```

Each object is described by a tuple which lists the name of the sheet on which the object is located, the type of object, the object's content or label (if applicable), the cell which the object is linked to (if applicable), and a list of pointers to the corresponding `GtkWidgets` (if any). Note that in general a GUI object may be associated with several widgets, as Gnumeric allows you to have multiple views on the same spreadsheet, so there will be one widget for each view an object is visible in. Also note that the content/label information depends on the particular type of object:

- List and combo widgets return the content link (referring to the cells in the spreadsheet holding the items shown in the list).
- Frame and button widgets return the label shown on the widget.
- Graphic objects like rectangles and ellipses return the text content of the object.
- Image objects (type `"image/xyz"`, where `xyz` is the type of image, such as `svg` or `png`) return a pointer to the image data in this field.

The `sheet_objects` function is a bit tricky to use, since some of the objects or their associated widgets might not have been created yet when the spreadsheet is loaded. Therefore it is necessary to use a trigger to make sure that the information is updated once all objects are fully displayed. The `pure_func.pure` script contains the following little wrapper around `sheet_objects` which does this:

```
pure_objects = trigger 0 (\_->all realized sheet_objects)
  (\_->matrix$map list sheet_objects) ()
with realized (_,_,_,_,w) = ~listp w || ~null w && ~any null w end;
```

See the `widgets.gnumeric` spreadsheet in the distribution for an example.

Possible uses of this facility are left to your imagination. Using Gnumeric's internal APIs and Pure's Gtk interface, you might manipulate the GUI widgets in various ways (add icons to buttons or custom child widgets to frames, etc.). One particularly useful case, for which Gnumeric/Pure has built-in support, is rendering an OpenGL scene in a Gnumeric frame widget, see below.

12.9.6 OpenGL Interface

Gnumeric/Pure provides special support for rendering OpenGL scenes into Gnumeric frame widgets. To actually use this, you must have Pure's OpenGL module installed. The following function is provided to equip a Gnumeric frame with the OpenGL rendering capability:

```
extern expr *pure_gl_window(char *name, int timeout,
                            expr *setup_cb, expr *config_cb,
                            expr *display_cb, expr *timer_cb,
                            expr *user_data) = gl_window;
```

The meaning of the parameters is as follows:

- `name` is a string which specifies the label of the frame widget into which the scene is to be rendered.
- `timeout` is a time value in milliseconds (an integer) which specifies the period for invocations of the `timer_cb` callback, see below. If this value is zero or negative then the timer callback is disabled.
- `setup_cb`, `config_cb`, `display_cb` and `timer_cb` are the Pure callback functions which are invoked by `gl_window` to actually render the scene. The callbacks are all invoked with two arguments: the `user_data` parameter that `gl_window` was invoked with, and a second parameter with callback-specific information as described below.
- `user_data` is any information that the caller wants to be passed as the first argument to the callback functions.

The different callbacks are:

- `setup_cb` is called for initializing the scene. It receives the `GtkDrawingArea` widget as the second argument. Typically this is used to set the initial projection and modelview matrices, enable lighting, etc.
- `config_cb` is called when the width or height of the frame widget changes so that the rendering parameters (typically the viewport) can be adjusted accordingly. It is invoked with a pair of integers `(w,h)` as the second argument, which denotes the new dimension allocated to the drawing area.

- `display_cb` is called whenever the contents of the drawing area needs to be redrawn. This typically does most of the work necessary to render the scene. The second callback argument is always `()`.
- `timer_cb` is called at regular intervals as specified with the `timeout` parameter (see above), unless the `timeout` value is zero or negative in which case this callback is disabled. The second callback argument is always `()`. The timer callback is typically used to incrementally adjust some parameters of the scene in order to render animations. When invoked, the callback automatically arranges for `display_cb` to be called afterwards, so you don't have to do that manually in your callback definition.

You'll need at least either the `display_cb` or the `timer_cb` function to render anything, but typically all of these callbacks will be needed for animated scenes. Callback functions which aren't needed can be specified as `()`.

There's also a related helper function which can be used as a trigger condition to defer rendering until the target frame widget has been realized:

```
extern bool pure_check_window(char *name) = check_window;
```

This function returns `true` as soon as the named frame widget is ready to go, at wich time `gl_windows` can be called on the widget. So your call to `gl_windows` should usually be wrapped up like this:

```
trigger 0 check_window
(\frame->gl_window frame timeout setup config display timer user_data) frame
```

Here's an example from `pure_glfunc.pure` which shows how these functions are to be used:

```
using pure_func, GL, GLU;
extern void gdk_gl_draw_teapot(bool solid, double scale);
gnm_info "gltest" = "sbfff";

gltest frame m a b c = trigger 0 check_window
  (\frame->gl_window frame (m*40) setup config display timer ()) frame
with
  setup _ _ = () when
    // Initialize.
    GL::ClearColor 0.1 0.1 0.3 1.0;
    GL::ShadeModel GL::SMOOTH;
    GL::Enable GL::DEPTH_TEST;
    // Initial projection and modelview matrices.
    GL::MatrixMode GL::PROJECTION;
    GL::LoadIdentity;
    GL::Rotatef 20.0 (-1.0) 0.0 0.0;
    GL::MatrixMode GL::MODELVIEW;
    GL::LoadIdentity;
    // Lighting.
    GL::Lightfv GL::LIGHT0 GL::DIFFUSE {1.0,0.0,0.0,1.0};
    GL::Lightfv GL::LIGHT0 GL::POSITION {2.0,2.0,-5.0,1.0};
    GL::Enable GL::LIGHTING;
    GL::Enable GL::LIGHT0;
```

```
  end;
  config _ (w,h) = GL::Viewport 0 0 w h;
  display _ _ = () when
    GL::Clear (GL::DEPTH_BUFFER_BIT or GL::COLOR_BUFFER_BIT);
    gdk_gl_draw_teapot true 0.5;
  end if m;
  display _ _ = () when
    GL::Clear (GL::DEPTH_BUFFER_BIT or GL::COLOR_BUFFER_BIT);
    GL::LoadIdentity;
    GL::Rotatef (scale 360 a) 0.0 1.0 0.0;
    GL::Rotatef (scale 360 b) 1.0 0.0 0.0;
    GL::Rotatef (scale 360 c) 0.0 0.0 1.0;
    gdk_gl_draw_teapot true 0.5;
  end;
  timer _ _ = () when
    GL::Rotatef (scale 36 a) 0.0 1.0 0.0;
    GL::Rotatef (scale 36 b) 1.0 0.0 0.0;
    GL::Rotatef (scale 36 c) 0.0 0.0 1.0;
  end;
  scale step x = (x/100*step);
end;
```

Have a look at the `gl-example.gnumeric` spreadsheet included in the distribution to see this example in action. (You first need to enable the "Pure OpenGL functions" in the Plugin Manager to make this work.) The screenshot below shows how the example looks like in Gnumeric.

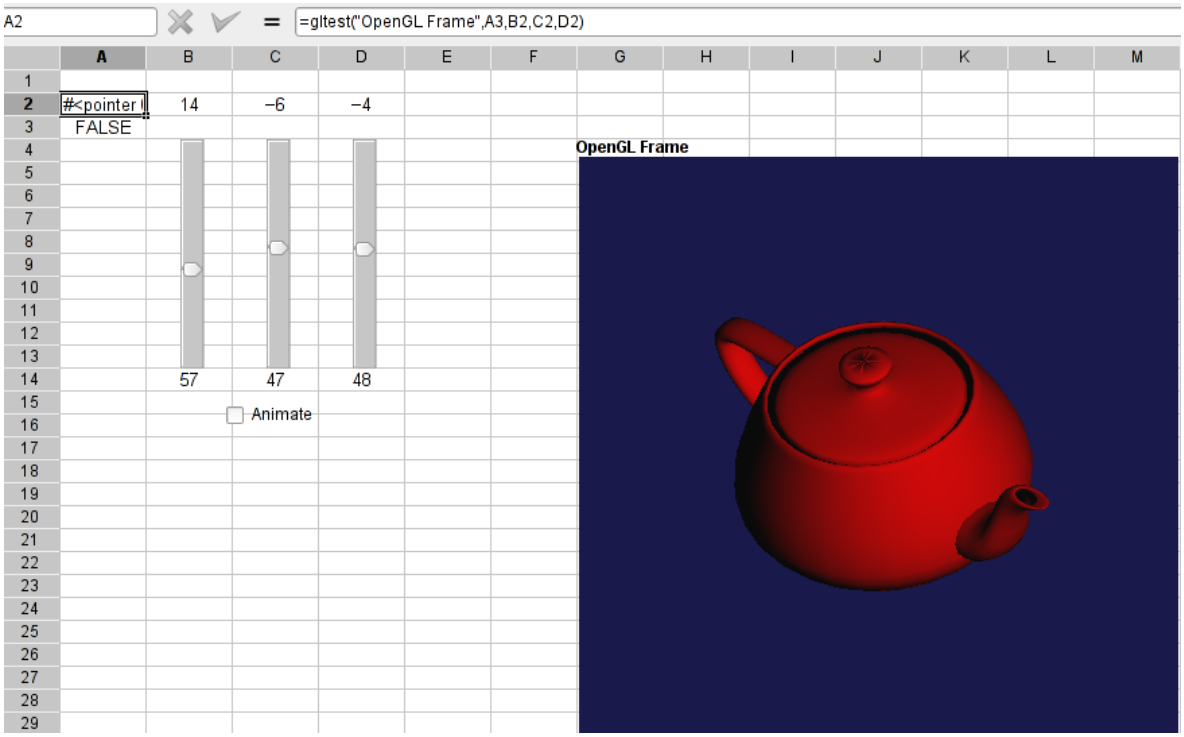

Figure 12.2: Gnumeric/Pure OpenGL example.

Chapter 13

Pure-GLPK - GLPK interface for the Pure programming language

Version 0.2, June 26, 2012

Jiri Spitz <jiri.spitz@bluetone.cz>

This module provides a feature complete GLPK interface for the Pure programming language, which lets you use all capabilities of the GNU Linear Programming Kit (GLPK) directly from Pure.

GLPK (see http://www.gnu.org/software/glpk) contains an efficient simplex LP solver, a simplex LP solver in exact arithmetics, an interior-point solver, a branch-and-cut solver for mixed integer programming and some specialized algorithms for net/grid problems. Using this interface you can build, modify and solve the problem, retrieve the solution, load and save the problem and solution data in standard formats and use any of advanced GLPK features.

The interface uses native Pure data types - lists and tuples - so that you need not perform any data conversions to/from GLPK internal data structures.

To make this module work, you must have a GLPK installation on your system, the version 4.42 or higher is required.

13.1 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-glpk-0.2.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure installed.

The default make options suppose that GLPK was configured with the following options: `--enable-dl --enable-odbc --enable-mysql --with-gmp --with-zlib`

Using the given options the depndencies are:

- GNU Multiprecision Library (GMP) - serves for the exact simplex solver. When disabled, the exact solver still works but it is much slower.
- ODBC library - serves for reading data directly from database tables within the GNU MathProg language translator through the ODBC interface.
- zlib compression library - enables reading and writing gzip compressed problem and solution files.
- MySQL client library - serves for reading data directly from MySQL tables within the GNU MathProg language translator.
- ltdl dlopen library - must be enabled together with any of ODBC, zlib or MySQL.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. The variable `ODBCLIB` specifies the ODBC library to be linked with. The default value is `ODBCLIB=-lodbc`. Please see the Makefile for details.

13.2 Error Handling

When an error condition occurs, the GLPK library itself prints an error mesage and terminates the application. This behaviour is not pleasant when working within an interpreter. Therefore, the Pure - GLPK bindings catches at least the most common errors like indices out of bounds. On such an error an appropriate message is returned to the interpreter. The less common errors are still trapped by the GLPK library.

When one of the most common errors occurs, an error term of the form `glp::error message` will be returned, which specifies what kind of error happend. For instance, an index out of boundsd will cause a report like the following:

```
glp::error "[Pure GLPK error] row index out of bounds"
```

You can check for such return values and take some appropriate action. By redefining `glp::error` accordingly, you can also have it generate exceptions or print an error message. For instance:

```
glp::error message = fprintf stderr "%s\n" message $$ ();
```

NOTE: When redefining `glp::error` in this manner, you should be aware that the return value of `glp::error` is what will be returned by the other operations of this module in case of an error condition. These return values are checked by other functions. Thus the return value should still indicate that an error has happened, and not be something that might be interpreted as a legal return value, such as an integer or a nonempty tuple. It is usually safe to have `glp::error` return an empty tuple or throw an exception, but other types of return values should be avoided.

IMPORTANT: It is really good to define a `glp::error` function, otherwise the errors might remain unnoticed.

13.3 Further Information and Examples

For further details about the operations provided by this module please see the GLPK Reference Manual. Sample scripts illustrating the usage of the module can be found in the examples directory.

13.4 Interface description

Most GLPK functions and symbols live in the namespace `glp`. There are a few functions and symbols in the namespace `lpx`. These functions and symbols are likely to be removed and replaced by new ones in the future.

In general, when you replace the `glp_` prefix from the GLPK Reference Manual with the namespace specification `glp::` then you receive the function name in this module. The same is valid for `lpx_` and `lpx::`. The symbolic constants are converted into lower case in this module, again obeying the same prefix rules.

13.5 Descriptions of interface functions

13.5.1 Basic API routines

Problem creating and modifying routines

Create the GLPK problem object **Synopsis**:

```
glp::create_prob
```

Parameters:

none

Returns:

pointer to the LP problem object

Example:

```
> let lp = glp::create_prob;
> lp;
#<pointer 0x9de7168>
```

Set the problem name **Synopsis**:

```
glp::set_prob_name lp name
```

Parameters:

lp pointer to the LP problem object

name problem name

Returns:

```
()
```

Example:

```
> glp::set_prob_name lp "Testing problem";
()
```

Set objective name **Synopsis**:

```
glp::set_obj_name lp name
```

Parameters:

lp pointer to the LP problem object

name objective name

Returns:

```
()
```

Example:

```
> glp::set_obj_name lp "Total costs";
()
```

Set the objective direction **Synopsis**:

```
glp::set_obj_dir lp direction
```

Parameters:

lp pointer to the LP problem object

direction one of the following:

glp::min minimize

glp::max maximize

Returns:

```
()
```

Example:

```
> glp::set_obj_dir lp glp::min;
()
```

Add new rows to the problem Synopsis:

```
glp::add_rows lp count
```

Parameters:

lp pointer to the LP problem object

count number of rows to add

Returns:

index of the first row added

Example:

```
> let first_added_row = glp_add_rows lp 3;
> first_added_row;
6
```

Add new columns to the problem Synopsis:

```
glp::add_cols lp count
```

Parameters:

lp pointer to the LP problem object

count number of columns to add

Returns:

index of the first column added

Example:

```
> let first_added_col = glp_add_cols lp 3;
> first_added_col;
5
```

Set the row name Synopsis:

```
glp::set_row_name lp (rowindex, rowname)
```

Parameters:

lp pointer to the LP problem object

rowindex row index

rowname row name

Returns:

()

Example:

```
> glp::set_row_name lp (3, "The third row");
()
```

Set the column name

Synopsis:

```
glp::set_col_name lp (colindex, colname)
```

Parameters:

lp pointer to the LP problem object

colindex column index

colname column name

Returns:

()

Example:

```
> glp::set_col_name lp (3, "The third column");
()
```

Set (change) row bounds

Synopsis:

```
glp::set_row_bnds lp (rowindex, rowtype, lowerbound, upperbound)
```

Parameters:

lp pointer to the LP problem object

rowindex row index

rowtype one of the following:

glp::fr free variable (both bounds are ignored)

glp::lo variable with lower bound (upper bound is ignored)

glp::up variable with upper bound (lower bound is ignored)

glp::db double bounded variable

glp::fx fixed variable (lower bound applies, upper bound is ignored)

lowerbound lower row bound

upperbound upper row bound

Returns: `()`

Example:: `glp::set_row_bnds lp (3, glp::up, 0.0, 150.0);`

Set (change) column bounds Synopsis:

```
glp::set_col_bnds lp (colindex, coltype, lowerbound, upperbound)
```

Parameters:

lp pointer to the LP problem object

colindex column index

coltype one of the following:

glp::fr free variable (both bounds are ignored)

glp::lo variable with lower bound (upper bound is ignored)

glp::up variable with upper bound (lower bound is ignored)

glp::db double bounded variable

glp::fx fixed variable (lower bound applies, upper bound is ignored)

lowerbound lower column bound

upperbound upper column bound

Returns:

`()`

Example:

```
> glp::set_col_bnds lp (3, glp::db, 100.0, 150.0);
()
```

Set (change) objective coefficient or constant term Synopsis:

```
glp::set_obj_coef lp (colindex, coefficient)
```

Parameters:

lp pointer to the LP problem object

colindex column index, zero index denotes the constant term (objective shift)

Returns:

```
()
```

Example:

```
> glp::set_obj_coef lp (3, 15.8);
()
```

Load or replace matrix row Synopsis:

```
glp::set_mat_row lp (rowindex, rowvector)
```

Parameters:

lp pointer to the LP problem object

rowindex row index

rowvector list of tuples (colindex, coefficient); only non-zero coefficients have to be specified, the order of column indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::set_mat_row lp (3, [(1, 3.0), (4, 5.2)]);
()
```

Load or replace matrix column Synopsis:

```
glp::set_mat_col lp (colindex, colvector)
```

Parameters:

lp pointer to the LP problem object

colindex column index

colvector list of tuples (rowindex, coefficient); only non-zero coefficients have to be specified, the order of row indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::set_mat_col lp (2, [(4, 2.0), (2, 1.5)]);
()
```

Load or replace the whole problem matrix

Synopsis:

```
glp::load_matrix lp matrix
```

Parameters:

lp pointer to the LP problem object

matrix list of tuples (rowindex, colindex, coefficient); only non-zero coefficients have to be specified, the order of indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::load_matrix lp [(1, 3, 5.0), (2, 2, 3.5), (3, 1, -2.0), (3, 2, 1.0)];
()
```

Check for duplicate elements in sparse matrix

Synopsis:

```
glp::check_dup numrows numcols indices
```

Parameters:

numrows number of rows

numcols number of columns

indices list of tuples (rowindex, colindex); indices of only non-zero coefficients have to be specified, the order of indices is not important

Returns:

returns one of the following:

0 the matrix has no duplicate elements

-k rowindex or colindex of the k-th element in indices is out of range

+k the k-th element in indices is duplicate

Remark:

Notice, that `k` counts from 1, whereas list members are counted from 0.

Example:

```
> glp::check_dup 3 3  [(1, 3), (2, 2), (3, 1), (2, 2)];
4
```

Sort elements of the constraint matrix **Synopsis**:

```
glp::sort_matrix lp
```

Parameters:

lp pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::sort_matrix lp;
()
```

Delete rows from the matrix **Synopsis**:

```
glp::del_rows lp rows
```

Parameters:

lp pointer to the LP problem object

rows list of indices of rows to be deleted; the order of indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Remark:

Deleting rows involves changing ordinal numbers of other rows remaining in the problem object. New ordinal numbers of the remaining rows are assigned under the assumption that the original order of rows is not changed.

Example:

```
> glp::del_rows lp [3, 4, 7];
()
```

Delete columns from the matrix **Synopsis**:

```
glp::del_cols lp cols
```

Parameters:

lp pointer to the LP problem object

cols list of indices of columns to be deleted; the order of indices is not important, duplicates are **not** allowed

Returns:

()

Remark:

Deleting columns involves changing ordinal numbers of other columns remaining in the problem object. New ordinal numbers of the remaining columns are assigned under the assumption that the original order of columns is not changed.

Example:

```
> glp::del_cols lp [6, 4, 5];
()
```

Copy the whole content of the GLPK problem object to another one

Synopsis:

```
glp::copy_prob destination source names
```

Parameters:

destination pointer to the destination LP problem object (must already exist)

source pointer to the source LP problem object

names one of the following:

glp::on copy all symbolic names as well

glp::off do not copy the symbolic names

Returns:

()

Example:

```
> glp::copy_prob lp_dest lp_src glp::on;
()
```

Erase all data from the GLPK problem object

Synopsis:

```
glp::erase_prob lp
```

Parameters:

lp pointer to the LP problem object, it remains still valid after the function call

Returns:

()

Example:

```
> glp::erase_prob lp;
()
```

Delete the GLPK problem object **Synopsis**:

```
glp::delete_prob lp
```

Parameters:

lp pointer to the LP problem object, it is not valid any more after the function call

Returns:

()

Example:

```
> glp::delete_prob lp;
()
```

Problem retrieving routines

Get the problem name **Synopsis**:

```
glp::get_prob_name lp
```

Parameters:

lp pointer to the LP problem object

Returns:

name of the problem

Example:

```
> glp::get_prob_name lp;
"Testing problem"
```

Get the objective name **Synopsis**:

```
glp::get_obj_name lp
```

Parameters:

lp pointer to the LP problem object

Returns:

name of the objective

Example:

```
> glp::get_obj_name lp;
"Total costs"
```

Get the objective direction **Synopsis**:

```
glp::get_obj_dir lp
```

Parameters:

lp pointer to the LP problem object

Returns:

returns one of the following:

glp::min minimize

glp::max maximize

Example:

```
> glp::get_obj_dir lp;
glp::min
```

Get number of rows **Synopsis**:

```
glp::get_num_rows lp
```

Parameters:

lp pointer to the LP problem object

Returns:

number of rows (constraints)

Example:

```
> glp::get_num_rows lp;
58
```

Get number of columns **Synopsis**:

```
glp::get_num_cols lp
```

Parameters:

lp pointer to the LP problem object

Returns:

number of columns (structural variables)

Example:

```
> glp::get_num_cols lp;
65
```

Get name of a row Synopsis:

```
glp::get_row_name lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

name of the given row

Example:

```
> glp::get_row_name lp 3;
"The third row"
```

Get name of a column Synopsis:

```
glp::get_col_name lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

name of the given column

Example:

```
> glp::get_col_name lp 2;
"The second column"
```

Get row type Synopsis:

```
glp::get_row_type lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

returns one of the following:

glp::fr free variable

glp::lo variable with lower bound

glp::up variable with upper bound

glp::db double bounded variable

glp::fx fixed variable

Example:

```
> glp::get_row_type lp 3;
glp::db
```

Get row lower bound

Synopsis:

```
glp::get_row_lb lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

the row lower bound; if the row has no lower bound then it returns the smallest double number

Example:

```
> glp::get_row_lb lp 3;
50.0
```

Get row upper bound

Synopsis:

```
glp::get_row_ub lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

the row upper bound; if the row has no upper bound then it returns the biggest double number

Example:

```
> glp::get_row_ub lp 3;
150.0
```

Get column type Synopsis:

```
glp::get_col_type lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

returns one of the following:

glp::fr free variable

glp::lo variable with lower bound

glp::up variable with upper bound

glp::db double bounded variable

glp::fx fixed variable

Example:

```
> glp::get_col_type lp 2;
glp::up
```

Get column lower bound Synopsis:

```
glp::get_col_lb lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

the column lower bound; if the column has no lower bound then it returns the smallest double number

Example:

```
> glp::get_col_lb lp 3;
-1.79769313486232e+308
```

Get column upper bound Synopsis:

```
glp::get_col_ub lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

the column upper bound; if the column has no upper bound then it returns the biggest double number

Example:

```
> glp::get_col_lb lp 3;
150.0
```

Get objective coefficient Synopsis:

```
glp::get_obj_coef lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index; zero index denotes the constant term (objective shift)

Returns:

the coefficient of given column in the objective

Example:

```
> glp::get_obj_coef lp 3;
5.8
```

Get number of nonzero coefficients Synopsis:

```
glp::get_num_nz lp
```

Parameters:

lp pointer to the LP problem object

Returns:

number of non-zero coefficients in the problem matrix

Example:

```
> glp::get_num_nz lp;
158
```

Retrive a row from the problem matrix Synopsis:

```
glp::get_mat_row lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

non-zero coefficients of the given row in a list form of tuples (colindex, coefficient)

Example:

```
> get_mat_row lp 3;
[(3,6.0),(2,2.0),(1,2.0)]
```

Retrive a column from the problem matrix Synopsis:

```
glp::get_mat_col lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

non-zero coefficients of the given column in a list form of tuples (rowindex, coefficient)

Example:

```
> get_mat_col lp 2;
[(3,2.0),(2,4.0),(1,1.0)]
```

Row and column searching routines

Create index for searching rows and columns by their names Synopsis:

```
glp::create_index lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::create_index lp;
()
```

Find a row number by name Synopsis:

```
glp::find_row lp rowname
```

Parameters:

lp pointer to the LP problem object

rowname row name

Returns:

ordinal number (index) of the row

Remark:

The search index is automatically created if it does not already exists.

Example:

```
> glp::find_row lp "The third row";
3
```

Find a column number by name Synopsis:

```
glp::find_col lp colname
```

Parameters:

lp pointer to the LP problem object

colname column name

Returns:

ordinal number (index) of the column

Remark:

The search index is automatically created if it does not already exists.

Example:

```
> glp::find_col lp "The second row";
2
```

Delete index for searching rows and columns by their names Synopsis:

```
glp::delete_index lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::delete:index lp;
()
```

Problem scaling routines

Set the row scale factor Synopsis:

```
glp::set_rii lp (rowindex, coefficient)
```

Parameters:

lp pointer to the LP problem object

rowindex row index

coefficient scaling coefficient

Returns:

()

Example:

```
> glp::set_rii lp (3, 258.6);
()
```

Set the column scale factor Synopsis:

```
glp::set_sjj lp (colindex, coefficient)
```

Parameters:

lp pointer to the LP problem object

colindex column index

coefficient scaling coefficient

Returns:

()

Example:

```
> glp::set_sjj lp (2, 12.8);
()
```

Retrieve the row scale factor Synopsis:

```
glp::get_rii lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

scaling coefficient of given row

Example:

```
> glp::get_rii lp 3;
258.6
```

Retrieve the column scale factor Synopsis:

```
glp::get_sjj lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

scaling coefficient of given column

Example:

```
> glp::get_sjj lp 2;
12.8
```

Scale the problem data according to supplied flags Synopsis:

```
glp::scale_prob lp flags
```

Parameters:

lp pointer to the LP problem object

flags symbolic integer constants which can be combined together by arithmetic **or**; the possible constants are:

glp::sf_gm perform geometric mean scaling

glp::sf_eq perform equilibration scaling

glp::sf_2n round scale factors to power of two

glp::sf_skip skip if problem is well scaled

glp::sf_auto choose scaling options automatically

Returns:

()

Example:

```
> glp::scale_prob lp (glp::sf_gm || glp::sf_2n);
()
```

Unscale the problem data

Synopsis:

```
glp::unscale_prob lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::unscale_prob lp;
()
```

LP basis constructing routines

Set the row status

Synopsis:

```
glp::set_row_stat lp (rowindex, status)
```

Parameters:

lp pointer to the LP problem object

rowindex row index

status one of the following:

glp::bs make the row basic (make the constraint inactive)

glp::nl make the row non-basic (make the constraint active)

glp::nu make the row non-basic and set it to the upper bound; if the row is not double-bounded, this status is equivalent to glp::nl (only in the case of this routine)

glp::nf the same as glp::nl (only in the case of this routine)

glp::ns the same as glp::nl (only in the case of this routine)

Returns:

()

Example:

```
> glp::set_row_stat lp (3, glp::nu);
()
```

Set the column status

Synopsis:

```
glp::set_col_stat lp (colindex, status)
```

Parameters:

lp pointer to the LP problem object

colindex column index

status one of the following:

glp::bs make the column basic

glp::nl make the column non-basic

glp::nu make the column non-basic and set it to the upper bound; if the column is not double-bounded, this status is equivalent to glp::nl (only in the case of this routine)

glp::nf the same as glp::nl (only in the case of this routine)

glp::ns the same as glp::nl (only in the case of this routine)

Returns:

()

Example:

```
> glp::set_col_stat lp (2, glp::bs);
()
```

Construct standard problem basis

Synopsis:

```
glp::std_basis lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::std_basis lp;
()
```

Construct advanced problem basis **Synopsis**:

```
glp::adv_basis lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::adv_basis lp;
()
```

Construct Bixby's problem basis **Synopsis**:

```
glp::cpx_basis lp
```

Parameters:

lp pointer to the LP problem object

Returns:

()

Example:

```
> glp::cpx_basis lp;
()
```

Simplex method routines

Solve the LP problem using simplex method **Synopsis**:

```
glp::simplex lp options
```

Parameters:

lp pointer to the LP problem object

options list of solver options in the form of tuples (option_name, value):

glp::msg_lev

(default: glp::msg_all) - message level for terminal output:

glp::msg_off: no output
glp::msg_err: error and warning messages only
glp::msg_on: normal output;
glp::msg_all: full output (including informational messages)

glp::meth (default: glp::primal) - simplex method option

glp::primal: use two-phase primal simplex
glp::dual: use two-phase dual simplex;
glp::dualp: use two-phase dual simplex, and if it fails, switch to the primal simplex

glp::pricing (default: glp::pt_pse) - pricing technique

glp::pt_std: standard (textbook)
glp::pt_pse: projected steepest edge

glp::r_test (default: glp::rt_har) - ratio test technique

glp::rt_std: standard (textbook)
glp::rt_har: Harris' two-pass ratio test

glp::tol_bnd (default: 1e-7) - tolerance used to check if the basic solution is primal feasible

glp::tol_dj (default: 1e-7) - tolerance used to check if the basic solution is dual feasible

glp::tol_piv (default: 1e-10) - tolerance used to choose eligble pivotal elements of the simplex table

glp::obj_ll (default: -DBL_MAX) - lower limit of the objective function - if the objective function reaches this limit and continues decreasing, the solver terminates the search - used in the dual simplex only

glp::obj_ul (default: +DBL_MAX) - upper limit of the objective function. If the objective function reaches this limit and continues increasing, the solver terminates the search - used in the dual simplex only

glp::it_lim (default: INT_MAX) - simplex iteration limit

glp::tm lim (default: INT_MAX) - searching time limit, in milliseconds

glp::out_frq (default: 200) - output frequency, in iterations - this parameter specifies how frequently the solver sends information about the solution process to the terminal

glp::out_dly (default: 0) - output delay, in milliseconds - this parameter specifies how long the solver should delay sending information about the solution process to the terminal

glp::presolve (default: glp::off) - LP presolver option:

glp::on: enable using the LP presolver
glp::off: disable using the LP presolver

Returns:

one of the following:

glp::ok the LP problem instance has been successfully solved; this code does not necessarily mean that the solver has found optimal solution, it only means that the solution process was successful

glp::ebadb unable to start the search, because the initial basis specified in the problem object is invalid - the number of basic (auxiliary and structural) variables is not the same as the number of rows in the problem object

glp::esing unable to start the search, because the basis matrix corresponding to the initial basis is singular within the working precision

glp::econd unable to start the search, because the basis matrix corresponding to the initial basis is ill-conditioned, i.e. its condition number is too large

glp::ebound unable to start the search, because some double-bounded (auxiliary or structural) variables have incorrect bounds

glp::efail the search was prematurely terminated due to the solver failure

glp::eobjll the search was prematurely terminated, because the objective function being maximized has reached its lower limit and continues decreasing (the dual simplex only)

glp::eobjul the search was prematurely terminated, because the objective function being minimized has reached its upper limit and continues increasing (the dual simplex only)

glp::eitlim the search was prematurely terminated, because the simplex iteration limit has been exceeded

glp::etmlim the search was prematurely terminated, because the time limit has been exceeded

glp::enopfs the LP problem instance has no primal feasible solution (only if the LP presolver is used)

glp::enodfs the LP problem instance has no dual feasible solution (only if the LP presolver is used)

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::simplex lp [(glp::presolve, glp::on), (glp::msg_lev, glp::msg_all)];
glp_simplex: original LP has 3 rows, 3 columns, 9 non-zeros
glp_simplex: presolved LP has 3 rows, 3 columns, 9 non-zeros
Scaling...
 A: min|aij| = 1,000e+000  max|aij| = 1,000e+001  ratio = 1,000e+001
Problem data seem to be well scaled
Crashing...
Size of triangular part = 3
*     0: obj =  0,000000000e+000  infeas = 0,000e+000 (0)
*     2: obj =  7,333333333e+002  infeas = 0,000e+000 (0)
OPTIMAL SOLUTION FOUND
glp::ok
```

Solve the LP problem using simplex method in exact arithmetics **Synopsis**:

```
glp::exact lp options
```

Parameters:

lp pointer to the LP problem object

options list of solver options in the form of tuples (option_name, value):

glp::it_lim (default: INT_MAX) - simplex iteration limit

glp::tm lim (default: INT_MAX) - searching time limit, in milliseconds

Returns:

one of the following:

glp::ok the LP problem instance has been successfully solved; this code does not necessarily mean that the solver has found optimal solution, it only means that the solution process was successful

glp::ebadb unable to start the search, because the initial basis specified in the problem object is invalid - the number of basic (auxiliary and structural) variables is not the same as the number of rows in the problem object

glp::esing unable to start the search, because the basis matrix corresponding to the initial basis is singular within the working precision

glp::ebound unable to start the search, because some double-bounded (auxiliary or structural) variables have incorrect bounds

glp::efail the search was prematurely terminated due to the solver failure

glp::eitlim the search was prematurely terminated, because the simplex iteration limit has been exceeded

glp::etmlim the search was prematurely terminated, because the time limit has been exceeded

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::exact lp [];
glp_exact: 3 rows, 3 columns, 9 non-zeros
GNU MP bignum library is being used
```

```
*     2:   objval =                      0   (0)
*     4:   objval =       733,333333333333   (0)
OPTIMAL SOLUTION FOUND
glp::ok
```

Retrieve generic status of basic solution **Synopsis**:

```
glp::get_status lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::undef solution is undefined

glp::feas solution is feasible

glp::infeas solution is infeasible

glp::nofeas no feasible solution exists

glp::opt solution is optimal

glp::unbnd solution is unbounded

Example:

```
> glp::get_status lp;
glp::opt
```

Retrieve generic status of primal solution **Synopsis**:

```
glp::get_prim_stat lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::undef primal solution is undefined

glp::feas primal solution is feasible

glp::infeas primal solution is infeasible

glp::nofeas no primal feasible solution exists

Example:

```
> glp::get_prim_stat lp;
glp::feas
```

Retrieve generic status of dual solution Synopsis:

```
glp::get_dual_stat lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::undef dual solution is undefined

glp::feas dual solution is feasible

glp::infeas dual solution is infeasible

glp::nofeas no dual feasible solution exists

Example:

```
> glp::get_dual_stat lp;
glp::feas
```

Retrieve value of the objective function Synopsis:

```
glp::get_obj_val lp
```

Parameters:

lp pointer to the LP problem object

Returns:

value of the objective function

Example:

```
> glp::get_obj_val lp
733.333333333333
```

Retrieve generic status of a row variable Synopsis:

```
glp::get_row_stat lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

one of the following:

glp::bs basic variable

glp::nl non-basic variable on its lower bound

glp::nu non-basic variable on its upper bound

glp::nf non-basic free (unbounded) variable

glp::ns non-basic fixed variable

Example:

```
> glp::get_row_stat lp 3;
glp::bs
```

Retrieve row primal value

Synopsis:: glp::get_row_prim lp rowindex

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

primal value of the row (auxiliary) variable

Example:

```
> glp::get_row_prim lp 3;
200.0
```

Retrieve row dual value

Synopsis:

```
glp::get_row_dual lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

dual value of the row (auxiliary) variable

Example:

```
> glp::get_row_dual lp 3;
0.0
```

Retrieve generic status of a column variable Synopsis:

```
glp::get_col_stat lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

one of the following:

glp::bs basic variable

glp::nl non-basic variable on its lower bound

glp::nu non-basic variable on its upper bound

glp::nf non-basic free (unbounded) variable

glp::ns non-basic fixed variable

Example:

```
> glp::get_col_stat lp 2;
glp::bs
```

Retrieve column primal value Synopsis:

```
glp::get_col_prim lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

primal value of the column (structural) variable

Example:

```
> glp::get_col_prim lp 2;
66.6666666666667
```

Retrieve column dual value **Synopsis**:

```
glp::get_col_dual lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

dual value of the column (structural) variable

Example:

```
> glp::get_col_dual lp 2;
0.0
```

Determine variable causing unboundedness **Synopsis**:

```
glp::get_unbnd_ray lp
```

Parameters:

lp pointer to the LP problem object

Returns:

The routine glp_get_unbnd_ray returns the number k of a variable, which causes primal or dual unboundedness. If 1 <= k <= m, it is k-th auxiliary variable, and if m + 1 <= k <= m + n, it is (k - m)-th structural variable, where m is the number of rows, n is the number of columns in the problem object. If such variable is not defined, the routine returns 0.

Remark:

If it is not exactly known which version of the simplex solver detected unboundedness, i.e. whether the unboundedness is primal or dual, it is sufficient to check the status of the variable with the routine glp::get_row_stat or glp::get_col_stat. If the variable is non-basic, the unboundedness is primal, otherwise, if the variable is basic, the unboundedness is dual (the latter case means that the problem has no primal feasible dolution).

Example:

```
> glp::get_unbnd_ray lp;
0
```

Interior-point method routines

Solve the LP problem using interior-point method **Synopsis**:

```
glp::interior lp options
```

Parameters:

lp pointer to the LP problem object

options list of solver options in the form of tuples (option_name, value):

glp::msg_lev

(default: glp::msg_all) - message level for terminal output:

glp::msg_off: no output
glp::msg_err: error and warning messages only
glp::msg_on: normal output;
glp::msg_all: full output (including informational messages)

glp::ord_alg (default: glp::ord_amd) - ordering algorithm option

glp::ord_none: use natural (original) ordering
glp::ord_qmd: quotient minimum degree (QMD)
glp::ord_amd: approximate minimum degree (AMD)
glp::ord_sysamd: approximate minimum degree (SYSAMD)

Returns:

one of the following:

glp::ok the LP problem instance has been successfully solved; this code does not necessarily mean that the solver has found optimal solution, it only means that the solution process was successful

glp::efail the problem has no rows/columns

glp::enocvg very slow convergence or divergence

glp::eitlim iteration limit exceeded

glp::einstab numerical instability on solving Newtonian system

Example:

```
> glp::interior lp [(glp::ord_alg, glp::ord_amd)];
Original LP has 3 row(s), 3 column(s), and 9 non-zero(s)
Working LP has 3 row(s), 6 column(s), and 12 non-zero(s)
Matrix A has 12 non-zeros
Matrix S = A*A' has 6 non-zeros (upper triangle)
Approximate minimum degree ordering (AMD)...
Computing Cholesky factorization S = L*L'...
Matrix L has 6 non-zeros
Guessing initial point...
Optimization begins...
  0: obj = -8,218489503e+002; rpi = 3,6e-001; rdi = 6,8e-001; gap = 2,5e-001
  1: obj = -6,719060895e+002; rpi = 3,6e-002; rdi = 1,9e-001; gap = 1,4e-002
  2: obj = -6,917210389e+002; rpi = 3,6e-003; rdi = 9,3e-002; gap = 3,0e-002
  3: obj = -7,267557732e+002; rpi = 2,1e-003; rdi = 9,3e-003; gap = 4,4e-002
  4: obj = -7,323038146e+002; rpi = 2,1e-004; rdi = 1,1e-003; gap = 4,8e-003
  5: obj = -7,332295932e+002; rpi = 2,1e-005; rdi = 1,1e-004; gap = 4,8e-004
  6: obj = -7,333229585e+002; rpi = 2,1e-006; rdi = 1,1e-005; gap = 4,8e-005
  7: obj = -7,333322959e+002; rpi = 2,1e-007; rdi = 1,1e-006; gap = 4,8e-006
  8: obj = -7,333332296e+002; rpi = 2,1e-008; rdi = 1,1e-007; gap = 4,8e-007
  9: obj = -7,333333230e+002; rpi = 2,1e-009; rdi = 1,1e-008; gap = 4,8e-008
 10: obj = -7,333333323e+002; rpi = 2,1e-010; rdi = 1,1e-009; gap = 4,8e-009
OPTIMAL SOLUTION FOUND
glp::ok
```

Retrieve status of interior-point solution **Synopsis**:

```
glp::ipt_status lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following

glp::undef interior-point solution is undefined

glp::opt interior-point solution is optimal

glp::infeas interior-point solution is infeasible

glp::nofeas no feasible primal-dual solution exists

Example:

```
> glp::ipt_status lp;
glp::opt
```

Retrieve the objective function value of interior-point solution **Synopsis**:

```
glp::ipt_obj_val lp
```

Parameters:

lp pointer to the LP problem object

Returns:

objective function value of interior-point solution

Example:

```
> glp::ipt_obj_val lp;
733.333332295849
```

Retrieve row primal value of interior-point solution

Synopsis:

```
glp::ipt_row_prim lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

primal value of the row (auxiliary) variable

Example:

```
> glp::ipt_row_prim lp 3;
200.000000920688
```

Retrieve row dual value of interior-point solution

Synopsis:

```
glp::ipt_row_dual lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

dual value of the row (auxiliary) variable

Example:

```
> glp::ipt_row_dual lp 3;
2.50607466186742e-008
```

Retrieve column primal value of interior-point solution Synopsis:

```
glp::ipt_col_prim lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

primal value of the column (structural) variable

Example:

```
> glp::ipt_col_prim lp 2;
66.666666406779
```

Retrieve column dual value of interior-point solution Synopsis:

```
glp::ipt_col_dual lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

dual value of the column (structural) variable

Example:

```
> glp::ipt_col_dual lp 2;
2.00019467655466e-009
```

Mixed integer programming routines

Set column kind Synopsis:

```
glp::set_col_kind lp (colindex, colkind)
```

Parameters:

lp pointer to the LP problem object

colindex column index

colkind column kind - one of the following:

glp::cv continuous variable

glp::iv integer variable

glp::bv binary variable

Returns:

()

Example:

```
> glp::set_col_kind lp (1, glp::iv);
()
```

Retrieve column kind Synopsis:

```
glp::get_col_kind lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

one of the following:

glp::cv continuous variable

glp::iv integer variable

glp::bv binary variable

Example:

```
> glp::get_col_kind lp 1;
glp::iv
```

Retrieve number of integer columns Synopsis:

```
glp::get_num_int lp
```

Parameters:

lp pointer to the LP problem object

Returns:

number of integer columns (including binary columns)

Example:

```
> glp_get_num_int lp;
1
```

Retrieve number of binary columns

Synopsis:

```
glp::get_num_bin lp
```

Parameters:

lp pointer to the LP problem object

Returns:

number of binary columns

Example:

```
> glp::get_num_bin lp
0
```

Solve the MIP problem using branch-and-cut method

Synopsis:

```
glp::intopt lp options
```

Parameters:

lp pointer to the LP problem object

options list of solver options in the form of tuples (option_name, value):

glp::msg_lev

(default: glp::msg_all) - message level for terminal output:

glp::msg_off: no output
glp::msg_err: error and warning messages only
glp::msg_on: normal output;
glp::msg_all: full output (including informational messages)

glp::br_tech (default: glp::bt::blb) - branching technique

glp::br_ffv: first fractional variable
glp::br_lfv: last fractional variable
glp::br_mfv: most fractional variable
glp::br_dth: heuristic by Driebeck and Tomlin
glp::br_pch: hybrid pseudocost heuristic

glp::bt_tech (default: glp::pt_pse) - backtracking technique

glp::bt_dfs: depth first search;
glp::bt_bfs: breadth first search;
glp::bt_blb: best local bound;
glp::bt_bph: best projection heuristic.

glp::pp_tech (default: glp::pp_all) - preprocessing technique

glp::pp_none: disable preprocessing;
glp::pp_root: perform preprocessing only on the root level
glp::pp_all: perform preprocessing on all levels

glp::fp_heur (default: glp::off) - feasibility pump heuristic:

glp::on: enable applying the feasibility pump heuristic
glp::off: disable applying the feasibility pump heuristic

glp::gmi_cuts

(default: glp::off) - Gomory's mixed integer cuts:

glp::on: enable generating Gomory's cuts;
glp::off: disable generating Gomory's cuts.

glp::mir_cuts

(default: glp::off) - mixed integer rounding (MIR) cuts:

glp::on: enable generating MIR cuts;
glp::off: disable generating MIR cuts.

glp::cov_cuts (default: glp::off) - mixed cover cuts:

glp::on: enable generating mixed cover cuts;
glp::off: disable generating mixed cover cuts.

glp::clq_cuts (default glp::off) - clique cuts:

glp::on: enable generating clique cuts;
glp::off: disable generating clique cuts.

glp::tol_int (default: 1e-5) - absolute tolerance used to check if optimal solution to the current LP relaxation is integer feasible

glp::tol_obj (default: 1e-7) - relative tolerance used to check if the objective value in optimal solution to the current LP relaxation is not better than in the best known integer feasible solution

glp::mip_gap (default: 0.0) - the relative mip gap tolerance; if the relative mip gap for currently known best integer feasible solution falls below this tolerance, the solver terminates the search - this allows obtainig suboptimal integer feasible solutions if solving the problem to optimality takes too long time

glp::tm lim (default: INT_MAX) - searching time limit, in milliseconds

glp::out_frq (default: 5000) - output frequency, in miliseconds - this parameter specifies how frequently the solver sends information about the solution process to the terminal

glp::out_dly (default: 10000) - output delay, in milliseconds - this parameter specifies how long the solver should delay sending information about the solution of the current LP relaxation with the simplex method to the terminal

glp::cb_func

(default: glp::off) - specifies whether to use the user-defined callback routine

glp::on: use user-defined callback function - the function `glp::mip_cb tree info` **must** be defined by the user

glp::off: do not use user-defined callback function

glp::cb_info (default: NULL) - transit pointer passed to the routine `glp::mip_cb tree info` (see above)

glp::cb_size (default: 0) - the number of extra (up to 256) bytes allocated for each node of the branch-and-bound tree to store application-specific data - on creating a node these bytes are initialized by binary zeros

glp::presolve (default: glp::off) - LP presolver option:

glp::on: enable using the MIP presolver

glp::off: disable using the MIP presolver

glp::binarize

(default: glp::off) - binarization (used only if the presolver is enabled):

glp::on: replace general integer variables by binary ones

glp::off: do not use binarization

Returns:

one of the following:

glp::ok the MIP problem instance has been successfully solved; this code does not necessarily mean that the solver has found optimal solution, it only means that the solution process was successful

glp::ebound unable to start the search, because some double-bounded (auxiliary or structural) variables have incorrect bounds or some integer variables have non-integer (fractional) bounds

glp::eroot unable to start the search, because optimal basis for initial LP relaxation is not provided - this code may appear only if the presolver is disabled

glp::enopfs unable to start the search, because LP relaxation of the MIP problem instance has no primal feasible solution - this code may appear only if the presolver is enabled

glp::enodfs unable to start the search, because LP relaxation of the MIP problem instance has no dual feasible solution; in other word, this code means that if the LP relaxation has at least one primal feasible solution, its optimal solution is unbounded, so if the MIP problem has at least one integer feasible solution, its (integer) optimal solution is also unbounded - this code may appear only if the presolver is enabled

glp::efail the search was prematurely terminated due to the solver failure

glp::emipgap the search was prematurely terminated, because the relative mip gap tolerance has been reached

glp::etmlim the search was prematurely terminated, because the time limit has been exceeded

glp::estop the search was prematurely terminated by application - this code may appear only if the advanced solver interface is used

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::intopt lp [(glp::presolve, glp::on)];
ipp_basic_tech:  0 row(s) and 0 column(s) removed
ipp_reduce_bnds: 2 pass(es) made, 3 bound(s) reduced
ipp_basic_tech:  0 row(s) and 0 column(s) removed
ipp_reduce_coef: 1 pass(es) made, 0 coefficient(s) reduced
glp_intopt: presolved MIP has 3 rows, 3 columns, 9 non-zeros
glp_intopt: 3 integer columns, none of which are binary
Scaling...
 A: min|aij| =  1,000e+00  max|aij| =  1,000e+01  ratio =  1,000e+01
Problem data seem to be well scaled
Crashing...
Size of triangular part = 3
Solving LP relaxation...
*     2: obj =   0,000000000e+00  infeas =  0,000e+00 (0)
*     5: obj =   7,333333333e+02  infeas =  0,000e+00 (0)
OPTIMAL SOLUTION FOUND
Integer optimization begins...
+     5: mip =     not found yet <=              +inf        (1; 0)
+     6: >>>>>   7,320000000e+02 <=   7,320000000e+02   0.0% (2; 0)
+     6: mip =   7,320000000e+02 <=     tree is empty   0.0% (0; 3)
INTEGER OPTIMAL SOLUTION FOUND
glp::ok
```

Retrieve status of mip solution **Synopsis**:

```
glp::mip_status lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::undef MIP solution is undefined

glp::opt MIP solution is integer optimal

glp::feas MIP solution is integer feasible, however, its optimality (or non-optimality) has not been proven, perhaps due to premature termination of the search

glp::nofeas problem has no integer feasible solution (proven by the solver)

Example:

```
> glp::mip_status lp;
glp::opt
```

Retrieve the objective function value of mip solution **Synopsis**:

```
glp::mip_obj_val lp
```

Parameters:

lp pointer to the LP problem object

Returns:

objective function value of mip solution

Example:

```
> glp::mip_obj_val lp;
732.0
```

Retrieve row value of mip solution **Synopsis**:

```
glp::mip_row_val lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

row value (value of auxiliary variable)

Example:

```
> glp::mip_row_val lp 3;
200.0
```

Retrieve column value of mip solution **Synopsis**:

```
glp::mip_col_val lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

column value (value of structural variable)

Example:

```
> glp::mip_col_val lp 2;
67.0
```

Additional routines

Check Karush-Kuhn-Tucker conditions **Synopsis**:

```
lpx::check_kkt lp scaled
```

Parameters:

lp pointer to the LP problem object

scaled one of the following:

true test the scaled problem

false test the unscaled problem

Returns:

list of four tuples with five mebers (see GLPK reference manual):

Condition	Member	Comment
(KKT.PE)	pe_ae_max	Largest absolute error
	pe_ae_row	Number of row with largest absolute error
	pe_re_max	Largest relative error
	pe_re_row	Number of row with largest relative error
	pe_quality	Quality of primal solution
(KKT.PB)	pb_ae_max	Largest absolute error
	pb_ae_ind	Number of variable with largest absolute error
	pb_re_max	Largest relative error
	pb_re_ind	Number of variable with largest relative error
	pb_quality	Quality of primal feasibility
(KKT.DE)	de_ae_max	Largest absolute error
	de_ae_col	Number of column with largest absolute error
	de_re_max	Largest relative error
	de_re_col	Number of column with largest relative error
	de_quality	Quality of dual solution
(KKT.DB)	db_ae_max	Largest absolute error
	db_ae_ind	Number of variable with largest absolute error
	db_re_max	Largest relative error
	db_re_ind	Number of variable with largest relative error
	db_quality	Quality of dual feasibility

where number of variable is (1 <= k <= m) for auxiliary variable and (m+1 <= k <= m+n) for structural variable

Example:

```
> lpx::check_kkt lp true;
[(1.4210854715202e-14,2,3.54385404369127e-17,3,"H"),(0.0,0,0.0,0,"H"),
(4.44089209850063e-16,1,2.11471052309554e-17,1,"H"),(0.0,0,0.0,0,"H")]
```

13.5.2 Utility API routines

Problem data reading/writing routines

Read LP problem data from a MPS file **Synopsis**:

```
glp::read_mps lp format filename
```

Parameters:

lp pointer to the LP problem object

format one of the following

glp::mps_deck fixed (ancient) MPS file format

glp::mps_file free (modern) MPS file format

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp::read_mps decompresses it "on the fly"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mps lp glp::mps_deck "examples/plan.mps";
Reading problem data from 'examples/plan.mps'...
Problem PLAN
Objective R0000000
8 rows, 7 columns, 55 non-zeros
63 records were read
0
```

Write LP problem data into a MPS file

Synopsis:

```
glp::write_mps lp format filename
```

Parameters:

lp pointer to the LP problem object

format one of the following

glp::mps_deck fixed (ancient) MPS file format

glp::mps_file free (modern) MPS file format

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp_write_mps performs automatic compression on writing it

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_mps lp glp::mps_file "examples/plan1.mps";
Writing problem data to 'examples/plan1.mps'...
63 records were written
0
```

Read LP problem data from a CPLEX file

Synopsis:

```
glp::read_lp lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp::read_lp decompresses it "on the fly"

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::read_lp lp "examples/plan.lp";
reading problem data from 'examples/plan.lp'...
8 rows, 7 columns, 48 non-zeros
39 lines were read
0
```

Write LP problem data into a CPLEX file

Synopsis:

```
glp::write_lp lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp::write_lp performs automatic compression on writing it

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_lp lp "examples/plan1.lp";
writing problem data to 'examples/plan1.lp'...
29 lines were written
0
```

Read LP problem data in GLPK format

Synopsis:

```
glp::read_prob lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp::read_prob decompresses it "on the fly"

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::read_prob lp "examples/plan.glpk";
reading problem data from 'examples/plan.glpk'...
8 rows, 7 columns, 48 non-zeros
86 lines were read
0
```

Write LP problem data in GLPK format **Synopsis**:

```
glp::write_prob lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name - if the file name ends with suffix **.gz**, the file is assumed to be compressed, in which case the routine glp::write_prob performs automatic compression on writing it

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_prob lp "examples/plan1.glpk";
writing problem data to 'examples/plan1.glpk'...
86 lines were written
0
```

Routines for MathProg models

Create the MathProg translator object **Synopsis**:

```
glp::mpl_alloc_wksp
```

Parameters:

none

Returns:

pointer to the MathProg translator object

Example:

```
> let mpt = glp::mpl_alloc_wksp;
> mpt;
#<pointer 0xa0d0180>
```

Read and translate model section Synopsis:

```
glp::mpl_read_model tranobject filename skip
```

Parameters:

tranobject pointer to the MathProg translator object

filename file name

skip if **0** then the data section from the model file is read; if non-zero, the data section in the data model is skipped

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> mpl_read_model mpt "examples/sudoku.mod" 1;
Reading model section from examples/sudoku.mod...
examples/sudoku.mod:69: warning: data section ignored
69 lines were read
0
```

Read and translate data section Synopsis:

```
glp::mpl_read_data tranobject filename
```

Parameters:

tranobject pointer to the MathProg translator object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::mpl_read_data mpt "examples/sudoku.dat";
Reading data section from examples/sudoku.dat...
16 lines were read
0
```

Generate the model Synopsis:

```
glp::mpl_generate tranobject filename
```

Parameters:

tranobject pointer to the MathProg translator object

filename file name

Returns:

0 if generating went OK; non-zero in case of an error

Example:

```
> glp::mpl_generate mpt "examples/sudoku.lst";
Generating fa...
Generating fb...
Generating fc...
Generating fd...
Generating fe...
Model has been successfully generated
0
```

Build problem instance from the model

Synopsis:

```
glp::mpl_build_prob tranobject lp
```

Parameters:

tranobject pointer to the MathProg translator object

lp pointer to the LP problem object

Returns:

`()`

Example:

```
> glp::mpl_build_prob mpt lp;
()
```

Postsolve the model

Synopsis:

```
glp::mpl_postsolve tran lp solution
```

Parameters:

tranobject pointer to the MathProg translator object

lp pointer to the LP problem object

solution one of the following:

glp::sol use the basic solution

glp::ipt use the interior-point solution

glp::mip use mixed integer solution

Returns:

0 if postsolve went OK; non-zero in case of an error

Example:

```
> glp::mpl_postsolve mpt lp glp::sol;
Model has been successfully processed
0
```

Delete the MathProg translator object **Synopsis**:

```
glp::mpl_free_wksp tranobject
```

Parameters:

tranobject pointer to the MathProg translator object

Returns:

`()`

Example:

```
> glp::mpl_free_wksp mpt;
()
```

Problem solution reading/writing routines

Write basic solution in printable format **Synopsis**:

```
glp::print_sol lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_sol lp "examples/test.txt";
Writing basic solution to 'examples/test.txt'...
0
```

Read basic solution from a text file **Synopsis**:

```
glp::read_sol lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_sol lp "examples/test.txt";
Reading basic solution from 'examples/test.txt'...
1235 lines were read
0
```

Write basic solution into a text file

Synopsis:

```
glp::write_sol lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_sol lp "examples/test.txt";
Writing basic solution to 'examples/test.txt'...
1235 lines were written
0
```

Print sensitivity analysis report

Synopsis:

```
glp::print_ranges lp indices filename
```

Parameters:

lp pointer to the LP problem object

indices list indices k of of rows and columns to be included in the report. If $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object. An empty lists means printing report for all rows and columns.

filename file name

Returns:

0 if the operation was successful

non-zero if the operation failed

Example:

```
> glp::print_ranges lp [] "sensitivity.rpt";
Write sensitivity analysis report to 'sensitivity.rpt'...
0
```

Write interior-point solution in printable format

Synopsis:

```
glp::print_ipt lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_ipt lp "examples/test.txt";
Writing interior-point solution to 'examples/test.txt'...
0
```

Read interior-point solution from a text file

Synopsis:

```
glp::read_ipt lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_ipt lp "examples/test.txt";
Reading interior-point solution from 'examples/test.txt'...
1235 lines were read
0
```

Write interior-point solution into a text file

Synopsis:

```
glp::write_ipt lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_ipt lp "examples/test.txt";
Writing interior-point solution to 'examples/test.txt'...
1235 lines were written
0
```

Write MIP solution in printable format

Synopsis:

```
glp::print_mip lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_mip lp "examples/test.txt";
Writing MIP solution to 'examples/test.txt'...
0
```

Read MIP solution from a text file

Synopsis:

```
glp::read_mip lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mip lp "examples/test.txt";
Reading MIP solution from 'examples/test.txt'...
1235 lines were read
0
```

Write MIP solution into a text file Synopsis:

```
glp::write_mip lp filename
```

Parameters:

lp pointer to the LP problem object

filename file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_mip lp "examples/test.txt";
Writing MIP solution to 'examples/test.txt'...
1235 lines were written
0
```

13.5.3 Advanced API routines

LP basis routines

Check whether basis factorization exists Synopsis:

```
glp::bf_exists lp
```

Parameters:

lp pointer to the LP problem object

Returns:

non-zero the basis factorization exists and can be used for calculations

0 the basis factorization does not exist

Example:

```
> glp::bf:exists lp;
1
```

Compute the basis factorization **Synopsis**:

```
glp::factorize lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::ok the basis factorization has been successfully computed

glp::ebadb the basis matrix is invalid, because the number of basic (auxiliary and structural) variables is not the same as the number of rows in the problem object

glp::esing the basis matrix is singular within the working precision

glp::exond the basis matrix is ill-conditioned, i.e. its condition number is too large

Example:

```
> glp::factorize lp;
glp::ok
```

Check whether basis factorization has been updated **Synopsis**:

```
glp::bf_updated lp
```

Parameters:

lp pointer to the LP problem object

Returns:

0 if the basis factorization has been just computed from "scratch"

non-zero if the factorization has been updated at least once

Example:

```
> glp::bf_updated lp;
0
```

Get basis factorization parameters **Synopsis**:

```
glp::get_bfcp lp
```

Parameters:

lp pointer to the LP problem object

Returns:

complete list of options in a form of tuples (option_name, value):

glp::fact_type basis factorization type:

glp::bf_ft LU + Forrest–Tomlin update

glp::bf_bg LU + Schur complement + Bartels–Golub update

glp::bf_gr LU + Schur complement + Givens rotation update

glp::lu_size the initial size of the Sparse Vector Area, in non-zeros, used on computing LU-factorization of the basis matrix for the first time - if this parameter is set to 0, the initial SVA size is determined automatically

glp::piv_tol threshold pivoting (Markowitz) tolerance, 0 < piv_tol < 1, used on computing LU-factorization of the basis matrix

glp::piv_lim this parameter is used on computing LU-factorization of the basis matrix and specifies how many pivot candidates needs to be considered on choosing a pivot element, piv_lim $\geq$ 1

glp::suhl this parameter is used on computing LU-factorization of the basis matrix

glp::on enables applying the heuristic proposed by Uwe Suhl

glp::off disables this heuristic

glp::eps_tol epsilon tolerance, eps_tol $\geq$ 0, used on computing LU-factorization of the basis matrix

glp::max_gro maximal growth of elements of factor U, max_gro $\geq$ 1, allowable on computing LU-factorization of the basis matrix

glp::nfs_max maximal number of additional row-like factors (entries of the eta file), nfs_max $\geq$ 1, which can be added to LU-factorization of the basis matrix on updating it with the Forrest–Tomlin technique

glp::upd_tol update tolerance, 0 < upd_tol < 1, used on updating LU-factorization of the basis matrix with the Forrest–Tomlin technique

glp::nrs_max maximal number of additional rows and columns, nrs_max $\geq$ 1, which can be added to LU-factorization of the basis matrix on updating it with the Schur complement technique

glp::rs_size the initial size of the Sparse Vector Area, in non-zeros, used to store non-zero elements of additional rows and columns introduced on updating LU-factorization of the basis matrix with

the Schur complement technique - if this parameter is set to 0, the initial SVA size is determined automatically

Example:

```
> glp::get_bfcp lp;
[(glp::fact_type,glp::bf_ft),(glp::lu_size,0),(glp::piv_tol,0.1),
(glp::piv_lim,4),(glp::suhl,glp::on),(glp::eps_tol,1e-15),
(glp::max_gro,10000000000.0),(glp::nfs_max,50),(glp::upd_tol,1e-06),
(glp::nrs_max,50),(glp::rs_size,0)]
```

Change basis factorization parameters

Synopsis:

```
glp::set_bfcp lp options
```

Parameters:

lp pointer to the LP problem object

options list of options in a form of tuples (option_name, value):

glp::fact_type (default: glp::bf_ft) - basis factorization type:

glp::bf_ft LU + Forrest–Tomlin update

glp::bf_bg LU + Schur complement + Bartels–Golub update

glp::bf_gr LU + Schur complement + Givens rotation update

glp::lu_size (default: 0) - the initial size of the Sparse Vector Area, in non-zeros, used on computing LU-factorization of the basis matrix for the first time - if this parameter is set to 0, the initial SVA size is determined automatically

glp::piv_tol (default: 0.10) - threshold pivoting (Markowitz) tolerance, 0 < piv_tol < 1, used on computing LU-factorization of the basis matrix.

glp::piv_lim (default: 4) - this parameter is used on computing LU-factorization of the basis matrix and specifies how many pivot candidates needs to be considered on choosing a pivot element, piv_lim ≥ 1

glp::suhl (default: glp::on) - this parameter is used on computing LU-factorization of the basis matrix.

glp::on enables applying the heuristic proposed by Uwe Suhl

glp::off disables this heuristic

glp::eps_tol (default: 1e-15) - epsilon tolerance, eps_tol ≥ 0, used on computing LU -factorization of the basis matrix.

glp::max_gro (default: 1e+10) - maximal growth of elements of factor U, max_gro ≥ 1, allowable on computing LU-factorization of the basis matrix.

glp::nfs_max (default: 50) - maximal number of additional row-like factors (entries of the eta file), nfs_max ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Forrest–Tomlin technique.

glp::upd_tol (default: 1e-6) - update tolerance, $0 < \text{upd\_tol} < 1$, used on updating LU -factorization of the basis matrix with the Forrest–Tomlin technique.

glp::nrs_max (default: 50) - maximal number of additional rows and columns, nrs_max ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Schur complement technique.

glp::rs_size (default: 0) - the initial size of the Sparse Vector Area, in non-zeros, used to store non-zero elements of additional rows and columns introduced on updating LU-factorization of the basis matrix with the Schur complement technique - if this parameter is set to 0, the initial SVA size is determined automatically

Remarks:

Options not mentioned in the option list are left unchanged.

All options will be reset to their default values when an empty option list is supplied.

Returns:

`()` if all options are OK, otherwise returns a list of bad options

Example:

```
> glp_set_bfcp lp [(glp::fact_type, glp::bf_ft), (glp::piv_tol, 0.15)];
()
```

Retrieve the basis header information Synopsis:

```
glp::get_bhead lp k
```

Parameters:

lp pointer to the LP problem object

k variable index in the basis matrix

Returns:

If basic variable (xB)k , $1 \leq k \leq m$, is i-th auxiliary variable ($1 \leq i \leq m$), the routine returns i. Otherwise, if (xB)k is j-th structural variable ($1 \leq j \leq n$), the routine returns m+j. Here m is the number of rows and n is the number of columns in the problem object.

Example:

```
> glp::get_bhead lp 3;
5
```

Retrieve row index in the basis header **Synopsis**:

```
glp::get_row_bind lp rowindex
```

Parameters:

lp pointer to the LP problem object

rowindex row index

Returns:

This routine returns the index k of basic variable (xB)k, $1 \leq k \leq m$, which is i-th auxiliary variable (that is, the auxiliary variable corresponding to i-th row), $1 \leq i \leq m$, in the current basis associated with the specified problem object, where m is the number of rows. However, if i-th auxiliary variable is non-basic, the routine returns zero.

Example:

```
> glp::get_row_bind lp 3;
1
```

Retrieve column index in the basis header **Synopsis**:

```
glp::get_col_bind lp colindex
```

Parameters:

lp pointer to the LP problem object

colindex column index

Returns:

This routine returns the index k of basic variable (xB)k, $1 \leq k \leq m$, which is j-th structural variable (that is, the structural variable corresponding to j-th column), $1 \leq j \leq n$, in the current basis associated with the specified problem object, where m is the number of rows, n is the number of columns. However, if j-th structural variable is non-basic, the routine returns zero.

Example:

```
> glp::get_col_bind lp 2;
3
```

Perform forward transformation Synopsis:

```
glp::ftran lp vector
```

Parameters:

lp pointer to the LP problem object

vector vector to be transformed - a dense vector in a form of a list of double numbers has to be supplied and the number of its members must exactly correspond to the number of LP problem constraints

Returns:

the transformed vector in the same format

Example:

```
> glp::ftran lp [1.5, 3.2, 4.8];
[1.8,0.466666666666667,-1.96666666666667]
```

Perform backward transformation Synopsis:

```
glp::btran lp vector
```

Parameters:

lp pointer to the LP problem object

vector vector to be transformed - a dense vector in a form of a list of double numbers has to be supplied and the number of its members must exactly correspond to the number of LP problem constraints

Returns:

the transformed vector in the same format

Example:

```
> glp::btran lp [1.5, 3.2, 4.8];
[-8.86666666666667,0.266666666666667,1.5]
```

Warm up LP basis Synopsis:

```
glp::warm_up lp
```

Parameters:

lp pointer to the LP problem object

Returns:

one of the following:

glp::ok the LP basis has been successfully "warmed up"

glp::ebadb the LP basis is invalid, because the number of basic variables is not the same as the number of rows

glp::esing the basis matrix is singular within the working precision

glp::econd the basis matrix is ill-conditioned, i.e. its condition number is too large

Example:

```
> glp::warm_up lp;
glp::e_ok
```

Simplex tableau routines

Compute row of the tableau **Synopsis**:

```
glp::eval_tab_row lp k
```

Parameters:

lp pointer to the LP problem object

k variable index such that it corresponds to some basic variable: if 1 $\leq$ k $\leq$ m, the basic variable is k-th auxiliary variable, and if m + 1 $\leq$ k $\leq$ m + n, the basic variable is (k $-$ m)-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

simplex tableau row in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::eval_tab_row lp 3;
[(1,2.0),(6,4.0)]
```

Compute column of the tableau **Synopsis**:

```
glp::eval_tab_col lp k
```

Parameters:

lp pointer to the LP problem object

k variable index such that it corresponds to some non-basic variable: if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

simplex tableau column in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::eval_tab_col lp 1;
[(3,2.0),(4,-0.666666666666667),(5,1.66666666666667)]
```

Transform explicitly specified row

Synopsis:

```
glp::transform_row lp rowvector
```

Parameters:

lp pointer to the LP problem object

rowvector row vector to be transformed in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

the transformed row in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::transform_row lp [(1, 3.0), (2, 3.5)];
[(1,3.83333333333333),(2,-0.0833333333333333),(6,-3.41666666666667)]
```

Transform explicitly specified column

Synopsis:

```
glp::transform_col lp colvector
```

Parameters:

lp pointer to the LP problem object

colvector column vector to be transformed in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th

auxiliary variable, and if m + 1 ≤ k ≤ m + n, the non-basic variable is (k − m)-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

the transformed column in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::transform_col lp [(2, 1.0), (3, 2.3)];
[(3,2.3),(4,-0.166666666666667),(5,0.166666666666667)]
```

Perform primal ratio test **Synopsis**:

```
glp::prim_rtest lp colvector dir eps
```

Parameters:

lp pointer to the LP problem object

colvector simplex tableau column in a sparse form as a list of tuples (k, value): if 1 ≤ k ≤ m, the basic variable is k-th auxiliary variable, and if m + 1 ≤ k ≤ m + n, the basic variable is (k − m)-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the primal solution must be feasible)

dir specifies in which direction the variable y changes on entering the basis: +1 means increasing, −1 means decreasing

eps relative tolerance (small positive number) used to skip small values in the column

Returns:

The routine returns the index, piv, in the colvector corresponding to the pivot element chosen, 1 ≤ piv ≤ len. If the adjacent basic solution is primal unbounded, and therefore the choice cannot be made, the routine returns zero.

Example:

```
> glp::prim_rtest lp [(3, 2.5), (5, 7.0)] 1 1.0e-5;
3
```

Perform dual ratio test **Synopsis**:

```
glp::dual_rtest lp rowvector dir eps
```

Parameters:

lp pointer to the LP problem object

rowvector simplex tableau row in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the dual solution must be feasible)

dir specifies in which direction the variable y changes on leaving the basis: +1 means increasing, −1 means decreasing

eps relative tolerance (small positive number) used to skip small values in the row

Returns:

The routine returns the index, piv, in the rowvector corresponding to the pivot element chosen, $1 \leq piv \leq len$. If the adjacent basic solution is dual unbounded, and therefore the choice cannot be made, the routine returns zero.

Example:

```
> glp::dual_rtest lp [(1, 1.5), (6, 4.0)] 1 1.0e-5;
6
```

Analyze active bound of non-basic variable

Synopsis:

```
glp::analyze_bound lp k
```

Parameters:

lp pointer to the LP problem object

k if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the solution must be optimal)

Returns:

The routine returns a tuple (limit1, var1, limit2 var2) where:

value1 the minimal value of the active bound, at which the basis still remains primal feasible and thus optimal. -DBL_MAX means that the active bound has no lower limit.

var1 the ordinal number of an auxiliary (1 to m) or structural (m + 1 to m + n) basic variable, which reaches its bound first and thereby limits further decreasing the active bound being analyzed. If value1 = -DBL_MAX, var1 is set to 0.

value2 the maximal value of the active bound, at which the basis still remains primal feasible and thus optimal. +DBL_MAX means that the active bound has no upper limit.

var2 the ordinal number of an auxiliary (1 to m) or structural (m + 1 to m + n) basic variable, which reaches its bound first and thereby limits further increasing the active bound being analyzed. If value2 = +DBL_MAX, var2 is set to 0.

Example:

```
> analyze_bound lp 2;
1995.06864446899,12,2014.03478832467,4
```

Analyze objective coefficient at basic variable

Synopsis:

```
glp::analyze_coef lp k
```

Parameters:

lp pointer to the LP problem object

k if $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the solution must be optimal)

Returns:

The routine returns a tuple (coef1, var1, value1, coef2 var2, value2) where:

coef1 the minimal value of the objective coefficient, at which the basis still remains dual feasible and thus optimal. -DBL_MAX means that the objective coefficient has no lower limit.

var1 is the ordinal number of an auxiliary (1 to m) or structural (m + 1 to m + n) non-basic variable, whose reduced cost reaches its zero bound first and thereby limits further decreasing the objective coefficient being analyzed. If coef1 = -DBL_MAX, var1 is set to 0.

value1 value of the basic variable being analyzed in an adjacent basis, which is defined as follows. Let the objective coefficient reaches its minimal value (coef1) and continues decreasing. Then the reduced cost of the limiting non-basic variable (var1) becomes dual infeasible and the current basis becomes non-optimal that forces the limiting non-basic variable to enter the basis replacing there some basic variable that leaves the basis to keep primal feasibility. Should note that on determining the adjacent basis current bounds of the basic variable being analyzed are ignored as if it were free (unbounded) variable, so it cannot leave the basis. It may happen

that no dual feasible adjacent basis exists, in which case value1 is set to -DBL_MAX or +DBL_MAX.

coef2 the maximal value of the objective coefficient, at which the basis still remains dual feasible and thus optimal. +DBL_MAX means that the objective coefficient has no upper limit.

var2 the ordinal number of an auxiliary (1 to m) or structural (m + 1 to m + n) non-basic variable, whose reduced cost reaches its zero bound first and thereby limits further increasing the objective coefficient being analyzed. If coef2 = +DBL_MAX, var2 is set to 0.

value2 value of the basic variable being analyzed in an adjacent basis, which is defined exactly in the same way as value1 above with exception that now the objective coefficient is increasing.

Example:

```
> analyze_coef lp 1;
-1.0,3,306.771624713959,1.79769313486232e+308,0,296.216606498195
```

13.5.4 Branch-and-cut API routines

All branch-and-cut API routines are supposed to be called from the callback routine. They cannot be called directly.

Basic routines

Determine reason for calling the callback routine

Synopsis:

```
glp::ios_reason tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

one of the following:

glp::irowgen request for row generation

glp::ibingo better integer solution found

glp::iheur request for heuristic solution

glp::icutgen request for cut generation

glp::ibranch request for branching

glp::iselect request for subproblem selection

glp::iprepro request for preprocessing

Example:

```
glp::ios:reason tree;
```

Access the problem object

Synopsis:

```
glp::ios_get_prob tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns a pointer to the problem object used by the MIP solver.

Example:

```
glp::ios_get_prob tree;
```

Determine additional row attributes

Synopsis:

```
glp::ios_row_attr tree rowindex
```

Parameters:

tree pointer to the branch-and-cut search tree

rowindex row index

Returns:

The routine returns a tuple consisting of three values (level, origin, klass):

level subproblem level at which the row was created

origin the row origin flag - one of the following:

glp::rf_reg regular constraint

glp::rf_lazy "lazy" constraint

glp::rf_cut cutting plane constraint

klass the row class descriptor, which is a number passed to the routine glp_ios_add_row as its third parameter - if the row is a cutting plane constraint generated by the solver, its class may be the following:

glp::rf_gmi Gomory's mixed integer cut

glp::rf_mir mixed integer rounding cut

glp::rf_cov mixed cover cut

glp::rf_clq clique cut

Example:

```
glp::ios_row_attr tree 3;
```

Compute relative MIP gap Synopsis:

```
glp::ios_mip_gap tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns the relative MIP gap.

Example:

```
> glp::ios_mip_gap tree;
```

Access application-specific data Synopsis:

```
glp::ios_node_data tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine glp_ios_node_data returns a pointer to the memory block for the specified subproblem. Note that if cb_size = 0 was specified in the call of the **intopt** function, the routine returns a null pointer.

Example:

```
> glp::ios_node_data tree 23;
```

Select subproblem to continue the search Synopsis:

```
glp::ios_select_node tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of the subproblem from which the search will continue

Returns:

()

Example:

```
> glp::ios_select_node tree 23;
```

Provide solution found by heuristic **Synopsis**:

```
glp::ios_heur_sol tree colvector
```

Parameters:

tree pointer to the branch-and-cut search tree

colvector solution found by a primal heuristic. Primal values of all variables (columns) found by the heuristic should be placed in the list, i. e. the list must contain n numbers where n is the number of columns in the original problem object. Note that the routine does not check primal feasibility of the solution provided.

Returns:

If the provided solution is accepted, the routine returns zero. Otherwise, if the provided solution is rejected, the routine returns non-zero.

Example:

```
> glp::ios_heur_sol tree [15.7, (-3.1), 2.2];
```

Check whether can branch upon specified variable **Synopsis**:

```
glp::ios_can_branch tree j
```

Parameters:

tree pointer to the branch-and-cut search tree

j variable (column) index

Returns:

The function returns non-zero if j-th variable can be used for branching. Otherwise, it returns zero.

Example:

```
> glp::ios_can_branch tree 23;
```

Choose variable to branch upon **Synopsis**:

```
glp::ios_branch_upon tree j selection
```

Parameters:

tree pointer to the branch-and-cut search tree

j ordinal number of the selected branching variable

selection one of the following:

glp::dn_brnch select down-branch

glp::up_brnch select up-branch

glp::no_brnch use general selection technique

Returns:

()

Example:

```
> glp::ios_branch_upon tree 23 glp::up_brnch;
```

Terminate the solution process

Synopsis:

```
glp::ios_terminate tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

()

Example:

```
> glp::ios_terminate tree;
```

The search tree exploring routines

Determine the search tree size

Synopsis:

```
glp::ios_tree_size tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns a tuple (a_cnt, n_cnt, t_cnt), where

a_cnt the current number of active nodes

n_cnt the current number of all (active and inactive) nodes

t_cnt the total number of nodes including those which have been already removed from the tree. This count is increased whenever a new node appears in the tree and never decreased.

Example:

```
> glp::ios_tree_size tree;
```

Determine current active subproblem

Synopsis:

```
glp::ios_curr_node tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns the reference number of the current active subproblem. If the current subproblem does not exist, the routine returns zero.

Example:

```
> glp::ios_curr_node tree;
```

Determine next active subproblem

Synopsis:

```
glp::ios_next_node tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of an active subproblem or zero

Returns:

If the parameter p is zero, the routine returns the reference number of the first active subproblem. If the tree is empty, zero is returned. If the parameter p is not zero, it must specify the reference number of some active subproblem, in which case the routine returns the reference number of the next active subproblem. If there is no next active subproblem in the list, zero is returned. All subproblems in the active list are ordered chronologically, i.e. subproblem A precedes subproblem B if A was created before B.

Example:

```
> glp::ios_next_node tree 23;
```

Determine previous active subproblem Synopsis:

```
glp::ios_prev_node tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of an active subproblem or zero

Returns:

If the parameter p is zero, the routine returns the reference number of the last active subproblem. If the tree is empty, zero is returned. If the parameter p is not zero, it must specify the reference number of some active subproblem, in which case the routine returns the reference number of the previous active subproblem. If there is no previous active subproblem in the list, zero is returned. All subproblems in the active list are ordered chronologically, i.e. subproblem A precedes subproblem B if A was created before B.

Example:

```
> glp::ios_prev_node tree 23;
```

Determine parent active subproblem Synopsis:

```
glp::ios_up_node tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of an active or inactive subproblem

Returns:

The routine returns the reference number of its parent subproblem. If the specified subproblem is the root of the tree, the routine returns zero.

Example:

```
> glp::ios_up_node tree 23;
```

Determine subproblem level Synopsis:

```
glp::ios_node_level tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of an active or inactive subproblem

Returns:

The routine returns the level of the given subproblem in the branch-and-bound tree. (The root subproblem has level 0.)

Example:

```
> glp::ios_node_level tree 23;
```

Determine subproblem local bound

Synopsis:

```
glp::ios_node_bound tree node
```

Parameters:

tree pointer to the branch-and-cut search tree

node reference number of an active or inactive subproblem

Returns:

The routine returns the local bound for the given subproblem.

Example:

```
> glp::ios_node_bound tree 23;
```

Find active subproblem with the best local bound

Synopsis:

```
glp::ios_best_node tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns the reference number of the active subproblem, whose local bound is best (i.e. smallest in case of minimization or largest in case of maximization). If the tree is empty, the routine returns zero.

Example:

```
> glp::ios_best_node tree;
```

The cut pool routines

Determine current size of the cut pool

Synopsis:

```
glp::ios_pool_size tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

The routine returns the current size of the cut pool, that is, the number of cutting plane constraints currently added to it.

Example:

```
> glp::ios_pool_size tree;
```

Add constraint to the cut pool

Synopsis:

```
glp::ios_add_row tree (name, klass, flags, row, rowtype, rhs)
```

Parameters:

tree pointer to the branch-and-cut search tree

name symbolic name of the constraint

klass specifies the constraint class, which must be either zero or a number in the range from 101 to 200. The application may use this attribute to distinguish between cutting plane constraints of different classes.

flags currently is not used and must be zero

row list of pairs (colindex, coefficient)

rowtype one of the following:

glp::lo $\sum$(aj.xj) $\geq$ RHS constraint

glp::up $\sum$(aj.xj) $\leq$ RHS constraint

rhs right hand side of the constraint

Returns:

The routine returns the ordinal number of the cutting plane constraint added, which is the new size of the cut pool.

Example:

```
> glp::ios_add_row tree ("new_constraint", 101, 0,
                         [(3, 15.0), (4, 6.7), (8, 1.25)], glp::up, 152.7);
```

Remove constraint from the cut pool

Synopsis:

```
glp::ios_del_row tree rowindex
```

Parameters:

tree pointer to the branch-and-cut search tree

rowindex index of row to be deleted from the cut pool

Returns:

()

Remark:

Note that deleting a constraint from the cut pool leads to changing ordinal numbers of other constraints remaining in the pool. New ordinal numbers of the remaining constraints are assigned under assumption that the original order of constraints is not changed.

Example:

```
> glp::ios_del_row tree 5;
```

Remove all constraints from the cut pool **Synopsis**:

```
glp::ios_clear_pool tree
```

Parameters:

tree pointer to the branch-and-cut search tree

Returns:

()

Example:

```
> glp::ios_clear_pool tree;
```

13.5.5 Graph and network API routines

Basic graph routines

Create the GLPK graph object **Synopsis**:

```
glp::create_graph v_size a_size
```

Parameters:

v_size size of vertex data blocks, in bytes, $0 \leq$ v size ≤ 256

a_size size of arc data blocks, in bytes, $0 \leq$ a size ≤ 256.

Returns:

The routine returns a pointer to the graph created.

Example:

```
> let g = glp::create_graph 32 64;
> g;
#<pointer 0x9de7168>
```

Set the graph name Synopsis:

```
glp::set_graph_name graph name
```

Parameters:

graph pointer to the graph object

name the graph name, an empty string erases the current name

Returns:

()

Example:

```
> glp::set_graph_name graph "MyGraph";
()
```

Add vertices to a graph Synopsis:

```
glp::add_vertices graph count
```

Parameters:

graph pointer to the graph object

count number of vertices to add

Returns:

The routine returns the ordinal number of the first new vertex added to the graph.

Example:

```
> glp::add_vertices graph 5;
18
```

Add arc to a graph Synopsis:

```
glp::add_arc graph i j
```

Parameters:

graph pointer to the graph object

i index of the tail vertex

j index of the head vertex

Returns:

()

Example:

```
> glp::add_arc graph 7 12;
()
```

Erase content of the GLPK graph object **Synopsis**:

```
glp::erase_graph graph v_size a_size
```

Parameters:

graph pointer to the graph object

v_size size of vertex data blocks, in bytes, $0 \leq$ v size ≤ 256

a_size size of arc data blocks, in bytes, $0 \leq$ a size ≤ 256.

Returns:

()

Remark:

The routine reinitialises the graph object. Its efect is equivalent to calling delete_graph followed by a call to create_graph.

Example:

```
> glp::erase_graph graph 16 34;
()
```

Delete the GLPK graph object **Synopsis**:

```
glp::delete_graph graph
```

Parameters:

graph pointer to the graph object

Returns:

()

Remark:

The routine destroys the graph object and invalidates the pointer. This is done automatically when the graph is not needed anymore, the routine need not be usually called.

Example:

```
> glp::delete_graph graph
()
```

Read graph in a plain text format **Synopsis**:

```
glp::read_graph graph filename
```

Parameters:

graph pointer to the graph object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_graph graph "graph_data.txt";
0
```

Write graph in a plain text format **Synopsis**:

```
glp::write_graph graph filename
```

Parameters:

graph pointer to the graph object

filename file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::write_graph graph "graph_data.txt";
0
```

Graph analysis routines

Find all weakly connected components of a graph **Synopsis**:

```
glp::weak_comp graph v_num
```

Parameters:

graph pointer to the graph object

v_num offset of the field of type int in the vertex data block, to which the routine stores the number of a weakly connected component containing that vertex - if v_num < 0, no component numbers are stored

Returns:

The routine returns the total number of components found.

Example:

```
> glp::weak_comp graph 16;
3
```

Find all strongly connected components of a graph Synopsis:

```
glp::strong_comp graph v_num
```

Parameters:

graph pointer to the graph object

v_num offset of the field of type int in the vertex data block, to which the routine stores the number of a strongly connected component containing that vertex - if v_num < 0, no component numbers are stored

Returns:

The routine returns the total number of components found.

Example:

```
> glp::strong_comp graph 16;
4
```

Minimum cost flow problem

Read minimum cost flow problem data in DIMACS format Synopsis:

```
glp::read_mincost graph v_rhs a_low a_cap a_cost filename
```

Parameters:

graph pointer to the graph object

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, the value is not stored

a_low offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, the lower bound is not stored

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0, the upper bound is not stored

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, the cost is not stored

fname the name of a text file to be read in - if the file name name ends with the suffix '.gz', the file is assumed to be compressed, in which case the routine decompresses it "on the fly"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mincost graph 0 8 16 24 "graphdata.txt";
0
```

Write minimum cost flow problem data in DIMACS format

Synopsis:

```
glp::write_mincost graph v_rhs a_low a_cap a_cost fname
```

Parameters:

graph pointer to the graph object

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, the value is not stored

a_low offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, the lower bound is not stored

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0, the upper bound is not stored

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, the cost is not stored

fname the name of a text file to be written out - if the file name name ends with the suffix '.gz', the file is assumed to be compressed, in which case the routine compresses it "on the fly"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::write_mincost graph 0 8 16 24 "graphdata.txt";
0
```

Convert minimum cost flow problem to LP Synopsis:

```
glp::mincost_lp lp graph names v_rhs a_low a_cap a_cost
```

Parameters:

lp pointer to the LP problem object

graph pointer to the graph object

names one of the following:

glp::on assign symbolic names of the graph object components to symbolic names of the LP problem object components

glp::off no symbolic names are assigned

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi = 0 for all nodes

a_low offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, it is assumed lij = 0 for all arcs

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij = 1 for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij = 0 for all arcs

Returns:

()

Example:

```
> glp::mincost_lp lp graph glp::on 0 8 16 24;
()
```

Solve minimum cost flow problem with out-of-kilter algorithm Synopsis:

```
glp::mincost_okalg graph v_rhs a_low a_cap a_cost a_x v_pi
```

Parameters:

graph pointer to the graph object

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi = 0 for all nodes

a_low offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, it is assumed lij = 0 for all arcs

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij = 1 for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij = 0 for all arcs

a_x offset of the field of type double in the arc data block, to which the routine stores xij, the arc flow found - if a_x < 0, the arc flow value is not stored

v_pi specifies an offset of the field of type double in the vertex data block, to which the routine stores pi, the node potential, which is the Lagrange multiplier for the corresponding flow conservation equality constraint

Remark:

Note that all solution components (the objective value, arc flows, and node potentials) computed by the routine are always integer-valued.

Returns:

The function returns a tuple in the form **(code, obj)**, where **code** is one of the following

glp::ok optimal solution found

glp::enopfs no (primal) feasible solution exists

glp::edata unable to start the search, because some problem data are either not integer-valued or out of range; this code is also returned if the total supply, which is the sum of bi over all source nodes (nodes with bi > 0), exceeds INT_MAX

glp::erange the search was prematurely terminated because of integer overflow

glp::efail an error has been detected in the program logic - if this code is returned for your problem instance, please report to <bug-glpk@gnu.org>

and **obj** is value of the objective function.

Example:

```
> glp::mincost_okalg graph 0 8 16 24 32 40;
(glp::ok, 15)
```

Klingman's network problem generator **Synopsis**:

```
glp::netgen graph v_rhs a_cap a_cost parameters
```

Parameters:

graph pointer to the graph object

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi = 0 for all nodes

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij = 1 for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij = 0 for all arcs

parameters tuple of exactly 15 integer numbers with the following meaning:

parm[1] iseed 8-digit positive random number seed

parm[2] nprob 8-digit problem id number

parm[3] nodes total number of nodes

parm[4] nsorc total number of source nodes (including transshipment nodes)

parm[5] nsink total number of sink nodes (including transshipment nodes)

parm[6] iarcs number of arc

parm[7] mincst minimum cost for arcs

parm[8] maxcst maximum cost for arcs

parm[9] itsup total supply

parm[10] ntsorc number of transshipment source nodes

parm[11] ntsink number of transshipment sink nodes

parm[12] iphic percentage of skeleton arcs to be given the maximum cost

parm[13] ipcap percentage of arcs to be capacitated

parm[14] mincap minimum upper bound for capacitated arcs

parm[15] maxcap maximum upper bound for capacitated arcs

Returns:

0 if the instance was successfully generated, nonzero otherwise

Example:

```
> glp::netgen graph 0 8 16 (12345678, 87654321, 20, 12, 8,
                            25, 5, 20, 300, 6, 5, 15, 100, 1, 30);
0
```

Grid-like network problem generator Synopsis:

```
glp::gridgen graph v_rhs a_cap a_cost parameters
```

Parameters:

graph pointer to the graph object

v_rhs offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi = 0 for all nodes

a_cap offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij = 1 for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij = 0 for all arcs

parameters tuple of exactly 14 integer numbers with the following meaning:

parm[1] two-ways arcs indicator:

1: if links in both direction should be generated

0: otherwise

parm[2] random number seed (a positive integer)

parm[3] number of nodes (the number of nodes generated might be slightly different to make the network a grid)

parm[4] grid width

parm[5] number of sources

parm[6] number of sinks

parm[7] average degree

parm[8] total flow

parm[9] distribution of arc costs:

1: uniform
2: exponential

parm[10] lower bound for arc cost (uniform), 100 lambda, (exponential)

parm[11] upper bound for arc cost (uniform), not used (exponential)

parm[12] distribution of arc capacities:

1: uniform
2: exponential

parm[13] lower bound for arc capacity (uniform), 100 lambda (exponential)

parm[14] upper bound for arc capacity (uniform), not used (exponential)

Returns:

0 if the instance was successfully generated, nonzero otherwise

Example:

```
> glp::gridgen graph 0 8 16 (1, 123, 20, 4, 7, 5, 3, 300, 1, 1, 5, 1, 5, 30);
0
```

Maximum flow problem

Read maximum cost flow problem data in DIMACS format

Synopsis:

```
glp::read_maxflow graph a_cap filename
```

Parameters:

graph pointer to the graph object

Returns:

Example:

```
>
```

Write maximum cost flow problem data in DIMACS format

Synopsis:

```
glp::write_maxflow graph s t a_cap filename
```

Parameters:

graph pointer to the graph object

Returns:

Example:

```
>
```

Convert maximum flow problem to LP

Synopsis:

```
glp::maxflow_lp lp graph names s t a_cap
```

Parameters:

graph pointer to the graph object

Returns:

Example:

```
>
```

Solve maximum flow problem with Ford-Fulkerson algorithm

Synopsis:

```
glp::maxflow_ffalg graph s t a_cap a_x v_cut
```

Parameters:

graph pointer to the graph object

Returns:

Example:

```
>
```

Goldfarb's maximum flow problem generator

Synopsis:

```
glp::rmfgen graph a_cap parameters
```

Parameters:

graph pointer to the graph object

Returns:

Example:

```
>
```

13.5.6 Miscellaneous routines

Library environment routines

Determine library version **Synopsis**:

```
'glp::version
```

Parameters:

none

Returns:

GLPK library version

Example:

```
> glp::version;
"4.38"
```

Enable/disable terminal output **Synopsis**:

```
glp::term_out switch
```

Parameters:

switch one of the following:

glp::on enable terminal output from GLPK routines

glp::off disable terminal output from GLPK routines

Returns:

()

Example:

```
> glp::term_out glp:off;
()
```

Enable/disable the terminal hook routine Synopsis:

```
glp::term_hook switch info
```

Parameters:

switch one of the following:

glp::on use the terminal callback function

glp::off don't use the terminal callback function

info pointer to a memory block which can be used for passing additional information to the terminal callback function

Returns:

()

Example:

```
> glp::term_hook glp::on NULL;
()
```

Get memory usage information Synopsis:

```
glp::mem_usage
```

Parameters:

none

Returns:

tuple consisting of four numbers:

- `count` (int) - the number of currently allocated memory blocks
- `cpeak` (int) - the peak value of `count` reached since the initialization of the GLPK library environment
- `total` (bigint) - the total amount, in bytes, of currently allocated memory blocks
- `tpeak` (bigint) - the peak value of `total` reached since the initialization of the GLPK library envirionment

Example:

```
> glp::mem_usage;
7,84,10172L,45304L
```

Set memory usage limit Synopsis:

```
glp::mem_limit limit
```

Parameters:

limit memory limit in megabytes

Returns:

()

Example:

```
> glp::mem_limit 200;
()
```

Free GLPK library environment

Synopsis:

```
glp::free_env
```

Parameters:

none

Returns:

()

Example:

```
> glp_free_env;
()
```

Chapter 14

Gnuplot bindings

Kay-Uwe Kirstein

14.1 Copying

Copyright (c) 2009, 2010 by Kay-Uwe Kirstein.

pure-gplot is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

14.2 Introduction

This module contains a pure binding to gnuplot. Communication to gnuplot is performed via pipes. The usual work flow to generate plot via gnuplot is the following:

1. open pipe via `open`
2. send plot commands, e.g., with `plot`
3. close pipe with `close`

14.3 Function Reference

14.3.1 Open / Closing Functions

```
gplot::open cmd;
```

opens a pipe to gnuplot, using *cmd*. *cmd* usually is something like `gnuplot` or `/path/to/gnuplot/bin/gnuplot` depending on your path configuration. `open` returns a pointer to the actual pipe for later usage, so a typical call to open might look like this:

```
let gp = gplot::open "/path_to_gnuplot/gnuplot";
```

gplot::GPLOT_EXE is a predefined variable with the standard Gnuplot executable. It is set to `pgnuplot` on Windows and to `gnuplot` otherwise and can be overridden bythe GPLOT_EXE environment variable. (`pgnuplot.exe` is a special executable for Windows, which is capable of stdin pipes in contrast to the normal `gnuplot.exe`). Usage of `gplot::GPLOT_EXE` might look like this:

```
let gp = gplot::open gplot::GPLOT_EXE;
```

```
gplot::close gp;
```

closes a gnuplot session, given by the handle *gp*.

14.3.2 Low-Level Commands

```
gplot::puts_no_echo string gp;
```

sends the string to the gnuplot session *gp* points to. As the name states, there is no echo read back from gnuplot (Don't know whether *gnuplot* or *pgnuplot.exe* supports reading/bidirectional pipes at all).

```
gplot::puts string gp;
```

is a convenience wrapper to `gplot::puts_no_echo`.

14.3.3 Plot Commands

The main (versatile) function to generate plots is the simple plot command, which expects a list of the data to be plotted.

```
gplot::plot gp data opt;
```

where *gp* is the pointer to the gnuplot session, *data* is a list containing the data to be plotted and *opt* is a tuple, containing options for the plot. *opt* might be empty () or `DEFAULT` for default options (refer to gnuplot for them).

If data for the x-axis (ordinate) should be explicitely given *plotxy* should be used instead:

```
gplot::plotxy_deprecated gp (xdata, ydata) opt;
```

```
gplot::plotxy gp (xdata, ydata) opt [];
```

Multiple datasets can be plotted into a single graph by combining them to tuples of lists:

```
gplot::plotxy gp (xdata, y1data, y2data, ..)  opt;
```

```
gplot::plotxy gp (xdata, y1data, y2data, ..)  opt [];
```

```
gplot::plotxy gp (xdata, y1data, y2data, ..)  opt titles;
```

where the latter form gives additional titles for each y-data set.

14.3.4 Plot Options

```
gplot::xtics gp list_of_tic_labels;
```

Sets the tic labels of the x-axis to the given text labels. The labels can be given aas a simple list of strings, which are taken as successive labels or as a list of tuples with the form `(value, label)`, in which case each label is placed at its `value` position.

`gplot::xtics gp ()` or `gplot::xtics gp "default";`

This restores the default tics on the y-axis.

```
gplot::title t;
```

Sets a title string on top of the plot (default location)

```
gplot::output gp terminal name;
```

Sets the terminal and output name for the successive plots. For some terminal additional options might be given:

`gplot::output gp (terminal, options) name`.

For terminals like x11 or windows, name can be empty `()`.

`gplot::xlabel gp name` or `gplot::ylabel gp name`

Adds labels to the x- or y-axis, respectively. An empty name removes the label for successive plots, e.g., `gplot::xlabel gp ""`.

14.3.5 Private Functions

`gpdata data`, `gpxydata (xdata, y1data, ..)`

Internal functions to handle lists of data point (`gpdata`) or tuples of lists of data points (`gpxydata`) and convert them to be understood by Gnuplot.

`gpxycmd`, `gpxycmdtitle`

Internal function to generate the plotting command for multiple datasets. `gpxycmdtitle` adds titles to each dataset, a.k.a plot legend.

`gplot::gpopt ("style", style, args);`

Internal function to convert a plot style to the respective gnuplot syntax

```
gplot::gptitle t;
```

Internal function to generate title information for individual datasets

Chapter 15

pure-gsl - GNU Scientific Library Interface for Pure

Version 0.11, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>
Eddie Rucker <erucker@bmc.edu>

License: GPL V3 or later, see the accompanying COPYING file

Building on Pure's GSL-compatible matrix support, this module aims to provide a complete wrapper for the GNU Scientific Library which provides a wide range of mathematical routines useful for scientific programming, number crunching and signal processing applications.

This is still work in progress, only a small part of the interface is finished right now. Here is a brief summary of the operations which are implemented:

- Matrix-scalar and matrix-matrix arithmetic. This is fairly complete and includes matrix multiplication, as well as element-wise exponentiation (^) and integer operations (div, mod, bit shifts and bitwise logical operations) which aren't actually in the GSL API.
- SVD (singular value decomposition), as well as the corresponding solvers, pseudo inverses and left and right matrix division. This is only available for real matrices right now, as GSL doesn't implement complex SVD.
- Random distributions (p.d.f. and c.d.f.) and statistic functions.
- Polynomial evaluation and roots.
- Linear least-squares fitting. Multi-fitting is not available yet.

Installation instructions: Get the latest source from http://pure-lang.googlecode.com/files/pure-gsl-0.11.tar.gz. Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure and GSL installed. The `make install` step is only necessary for system-wide installation.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. Please see the Makefile for details.

The current release requires GSL 1.11 or later and Pure 0.45 or later. Older GSL versions might still work, but then some operations may be missing. The latest and greatest GSL version is always available from http://www.gnu.org/software/gsl.

After installation, you can import the entire GSL interface as follows:

```
using gsl;
```

For convenience, the different parts of the GSL interface are also available as separate modules. E.g., if you only need the matrix operations:

```
using gsl::matrix;
```

In either case, the global `gsl_version` variable reports the installed GSL version:

```
> show gsl_version
let gsl_version = "1.11";
```

(This variable used to be defined by the Pure runtime but has been moved into pure-gsl as of Pure 0.37.)

Most other operations are declared in separate namespaces which are in 1-1 correspondence with the module names. Thus, e.g., the `gsl_poly_eval` routine is named `gsl::poly::eval` in Pure and can be found in the `gsl::poly` module and namespace. The `using namespace` declaration can be used to facilitate access to the operations in a given namespace, e.g.:

```
> using gsl::poly;
> using namespace gsl::poly;
> eval {1,2,3} 2;
17
```

See the `examples` folder in the sources for some examples.

If you'd like to contribute, please mail the authors or contact us at http://groups.google.com/group/pure-lang.

15.1 Polynomials

This module provides Pure wrappers for the GSL polynomial routines. For detail about the routines, see Chapter 6 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Polynomials.html.

Polynomials are represented by vectors (one row matrices).

15.1.1 Routines

`gsl::poly::`**`eval`** c::matrix x
: implements `gsl_poly_eval`, `gsl_poly_complex_eval`, and `gsl_complex_poly_eval` without the `len` parameter.

  GSL does not supply an integer routine for evaluating polynomials with `int` or `bigint` coefficients. Therefore, an integer routine has been provided in pure-gsl using the Chinese Remainder Theorem.

`gsl::poly::`**`dd_init`** x::matrix y::matrix
: implements `gsl_poly_dd_init` without the `size` parameter.

`gsl::poly::`**`dd_eval`** dd::matrix xa::matrix x::double
: implements `gsl_poly_dd_eval` without the `size` parameter.

`gsl::poly::`**`dd_taylor`** xp::double dd::matrix xa::matrix
: implements `gsl_poly_dd_taylor` without the `size` and workspace `w` arguments.

`gsl::poly::`**`solve_quadratic`** a b c
: implements `gsl_poly_solve_quadratic`. This function returns a list of roots instead of passing them through the parameters `x0` and `x1`.

`gsl::poly::`**`complex_solve_quadratic`** a b c
: implements `gsl_poly_complex_solve_quadratic`. This function returns a list of roots instead of passing trhough the parameters `z0` and `z1`.

`gsl::poly::`**`solve_cubic`** a b c
: implements `gsl_poly_solve_cubic`. This function returns a list of roots instead of passing them through the parameters `x0`, `x1`, and `x2`.

`gsl::poly::`**`complex_solve_cubic`** a b c
: implements `gsl_poly_complex_colve_cubic`. This function returns a list of roots instead of passing them through the parameters `z0`, `z1`, and `z2`.

`gsl::poly::`**`complex_solve`** c::matrix
: implements `gsl_poly_complex_solve` omitting the parametrs `n` and `w`. The GSL routines for creating and freeing the workspace are handled automatically.

15.1.2 Examples

Usage of each library routine is illustrated below.

```
> using gsl::poly;
> using namespace gsl::poly;
> eval {1,2,3} 2;
```

```
17
> eval {1.0,2.0,3.0} (-2.0);
9.0
> eval {1, 2, 2} (1+:1);
3.0+:6.0
> eval {1+:2, 2+:3, 2+:3} (1+:1);
-6.0+:11.0
> let dd = dd_init {1,2,3} {2,4,6};
> dd;
{2.0,2.0,0.0}
> dd_eval dd {1,2,3} 2;
4.0
> dd_taylor 0.0 dd {1,2,3};
{0.0,2.0,0.0}
> solve_quadratic 2 4 1;
[-1.70710678118655,-0.292893218813452]
> solve_quadratic 1 4 4;
[-2.0,-2.0]
> solve_quadratic 0 2 1;
[-0.5]
> solve_quadratic 1 2 8;
[]
> complex_solve_quadratic 0 2 1;
[-0.5+:0.0]
> complex_solve_quadratic 2 2 3;
[-0.5+:-1.11803398874989,-0.5+:1.11803398874989]
> solve_cubic 3 3 1;
[-1.0,-1.0,-1.0]
> solve_cubic 3 2 1;
[-2.32471795724475]
> complex_solve_cubic 2 2 1;
[-1.0+:0.0,-0.5+:-0.866025403784439,-0.5+:0.866025403784439]
> complex_solve {6,1,-7,-1,1};
[1.0+:0.0,-1.0+:0.0,-2.0+:0.0,3.0+:0.0]
```

15.2 Special Functions

This module is loaded via the command `using gsl::sf` and provides Pure wrappers for the GSL Special Functions. For details, see Chapter 7 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Special-Functions.html.

To load the library, use the Pure command `using gsl::sf`. Modes for the functions must be one of:

```
GSL_PREC_DOUBLE
GSL_PREC_SINGLE
GSL_PREC_APPROX
```

Results for some of the functions are returned as a Pure list instead of the `gsl_sf_result` or

`gsl_sf_result_e10` structures in C. In these cases, the resulting list is one of the following forms.

- `[val, err]` for the `gsl_sf_result` struct and
- `[val, err, e10]` for the `gsl_sf_result_e10` struct.

15.2.1 Airy Functions

`gsl::sf::`**`airy_Ai`** x
`gsl::sf::`**`airy_Ai`** (x, mode::int)
: implements `gsl_sf_airy_Ai`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_e`** x
`gsl::sf::`**`airy_Ai_e`** (x, mode::int)
: implements `gsl_sf_airy_Ai_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_scaled`** x
`gsl::sf::`**`airy_Ai_scaled`** (x, mode::int)
: implements `gsl_sf_airy_Ai_scaled`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_scaled_e`** x
`gsl::sf::`**`airy_Ai_scaled_e`** (x, mode::int)
: implements `gsl_sf_airy_Ai_scaled_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi`** x
`gsl::sf::`**`airy_Bi`** (x, mode::int)
: implements `gsl_sf_airy_Bi`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_e`** x
`gsl::sf::`**`airy_Bi_e`** (x, mode::int)
: implements `gsl_sf_airy_Bi_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_scaled`** x
`gsl::sf::`**`airy_Bi_scaled`** (x, mode::int)
: implements `gsl_sf_airy_Bi_scaled`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_scaled_e`** x
`gsl::sf::`**`airy_Bi_scaled_e`** (x, mode::int)
: implements `gsl_sf_airy_Bi_scaled_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_deriv`** x

`gsl::sf::`**`airy_Ai_deriv`** (x, mode::int)
implements `gsl_sf_airy_Ai_deriv`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_deriv_e`** x
`gsl::sf::`**`airy_Ai_deriv_e`** (x, mode::int)
implements `gsl_sf_airy_Ai_deriv_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_deriv_scaled`** x
`gsl::sf::`**`airy_Ai_deriv_scaled`** (x, mode::int)
implements `gsl_sf_airy_Ai_deriv_scaled`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Ai_deriv_scaled_e`** x
`gsl::sf::`**`airy_Ai_deriv_scaled_e`** (x, mode::int)
implements `gsl_sf_airy_Ai_deriv_scaled_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_deriv`** x
`gsl::sf::`**`airy_Bi_deriv`** (x, mode::int)
implements `gsl_sf_airy_Bi_deriv`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_deriv_e`** x
`gsl::sf::`**`airy_Bi_deriv_e`** (x, mode::int)
implements `gsl_sf_airy_Bi_deriv_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_deriv_scaled`** x
`gsl::sf::`**`airy_Bi_deriv_scaled`** (x, mode::int)
implements `gsl_sf_airy_Bi_deriv_scaled`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_Bi_deriv_scaled_e`** x
`gsl::sf::`**`airy_Bi_deriv_scaled_e`** (x, mode::int)
implements `gsl_sf_airy_Bi_deriv_scaled_e`. The first form computes the function with `mode = GSL_PREC_DOUBLE`.

`gsl::sf::`**`airy_zero_Ai`** s
implements `gsl_sf_airy_zero_Ai`.

`gsl::sf::`**`airy_zero_Ai_e`** s
implements `gsl_sf_airy_zero_Ai_e`.

`gsl::sf::`**`airy_zero_Bi`** s
implements `gsl_sf_airy_zero_Bi`.

`gsl::sf::`**`airy_zero_Bi_e`** s
implements `gsl_sf_airy_zero_Bi_e`.

`gsl::sf::`**`airy_zero_Ai_deriv`** s
implements `gsl_sf_airy_zero_Ai_deriv`.

gsl::sf::**airy_zero_Ai_deriv_e** s
    implements gsl_sf_airy_zero_Ai_deriv_e.

gsl::sf::**airy_zero_Bi_deriv** s
    implements gsl_sf_airy_zero_Bi_deriv.

gsl::sf::**airy_zero_Bi_deriv_e** s
    implements gsl_sf_airy_zero_Bi_deriv_e.

15.2.2 Examples

The following illustrate the Airy functions.

```
> using gsl::sf;
> using namespace gsl::sf;
> airy_Ai (-1.2); // defaults to GSL_PREC_DOUBLE
0.52619437480212
> airy_Ai_scaled (-1.2);
0.52619437480212
> airy_Ai (-1.2,GSL_PREC_APPROX);
0.526194374771687
> airy_Ai_scaled (-1.2, GSL_PREC_SINGLE);
0.526194374771687
> airy_Ai_e (-1.2);
[0.52619437480212,1.88330586480371e-15]
> airy_Ai_e (-1.2,GSL_PREC_APPROX);
[0.526194374771687,1.01942940819652e-08]
> airy_Ai_scaled_e (-1.2);
[0.52619437480212,1.88330586480371e-15]
> airy_Ai_scaled_e (-1.2,GSL_PREC_APPROX);
[0.526194374771687,1.01942940819652e-08]
> airy_Bi (-1.2);
-0.015821370184632
> airy_Bi_scaled (-1.2);
-0.015821370184632
> airy_Bi (-1.2,GSL_PREC_APPROX);
-0.0158213701898015
> airy_Bi_scaled (-1.2, GSL_PREC_SINGLE);
-0.0158213701898015
> airy_Bi_e (-1.2);
[-0.015821370184632,1.31448899295896e-16]
> airy_Bi_e (-1.2,GSL_PREC_APPROX);
[-0.0158213701898015,4.10638404843775e-10]
> airy_Bi_scaled_e (-1.2);
[-0.015821370184632,1.31448899295896e-16]
> airy_Bi_scaled_e (-1.2,GSL_PREC_APPROX);
[-0.0158213701898015,4.10638404843775e-10]
> airy_Ai_deriv (-1.2); // defaults to GSL_PREC_DOUBLE
0.107031569272281
> airy_Ai_deriv_scaled (-1.2);
0.107031569272281
```

```
> airy_Ai_deriv (-1.2,GSL_PREC_APPROX);
0.107031569264504
> airy_Ai_deriv_scaled (-1.2, GSL_PREC_SINGLE);
0.107031569264504
> airy_Ai_deriv_e (-1.2);
[0.107031569272281,3.02919983680384e-16]
> airy_Ai_deriv_e (-1.2,GSL_PREC_APPROX);
[0.107031569264504,9.25921017197604e-11]
> airy_Ai_deriv_scaled_e (-1.2);
[0.107031569272281,3.02919983680384e-16]
> airy_Ai_deriv_scaled_e (-1.2,GSL_PREC_APPROX);
[0.107031569264504,9.25921017197604e-11]
> airy_Bi_deriv (-1.2);
0.601710157437464
> airy_Bi_deriv_scaled (-1.2);
0.601710157437464
> airy_Bi_deriv (-1.2,GSL_PREC_APPROX);
0.601710157441937
> airy_Bi_deriv_scaled (-1.2, GSL_PREC_SINGLE);
0.601710157441937
> airy_Bi_deriv_e (-1.2);
[0.601710157437464,1.7029557943563e-15]
> airy_Bi_deriv_e (-1.2,GSL_PREC_APPROX);
[0.601710157441937,5.20534347823991e-10]
> airy_Bi_deriv_scaled_e (-1.2);
[0.601710157437464,1.7029557943563e-15]
> airy_Bi_deriv_scaled_e (-1.2,GSL_PREC_APPROX);
[0.601710157441937,5.20534347823991e-10]
> airy_zero_Ai 2;
-4.08794944413097
> airy_zero_Ai_e 3;
[-5.52055982809555,1.22581052599448e-15]
> airy_zero_Bi 2;
-3.27109330283635
> airy_zero_Bi_e 3;
[-4.83073784166202,1.07263927554824e-15]
> airy_zero_Ai_deriv 2;
-3.24819758217984
> airy_zero_Ai_deriv_e 3;
[-4.82009921117874,1.07027702504564e-15]
> airy_zero_Bi_deriv 2;
-4.07315508907183
> airy_zero_Bi_deriv_e 3;
[-5.5123957296636,1.22399773198358e-15]
```

15.2.3 Bessel Functions

`gsl::sf::`**`bessel_J0`** x
: implements `gsl_sf_bessel_J0`.

`gsl::sf::`**`bessel_J0_e`** x

implements gsl_sf_besselJ0_e.

gsl::sf::**bessel_J1** x
implements gsl_sf_bessel_J1.

gsl::sf::**bessel_J1_e** x
implements gsl_sf_bessel_J1_e.

gsl::sf::**bessel_Jn** n x
implements gsl_sf_bessel_Jn.

gsl::sf::**bessel_Jn_e** n x
implements gsl_sf_bessel_Jn_e.

gsl::sf::**bessel_Jn_array** nmin::int nmax::int x
implements gsl_sf_bessel_Jn_array.

gsl::sf::**bessel_Y0** x
implements gsl_sf_bessel_Y0.

gsl::sf::**bessel_Y0_e** x
implements gsl_sf_bessel_Y0_e.

gsl::sf::**bessel_Y1** x
implements gsl_sf_bessel_Y1.

gsl::sf::**bessel_Y1_e** x
implements gsl_sf_bessel_Y1_e.

gsl::sf::**bessel_Yn** x
implements gsl_sf_bessel_Yn.

gsl::sf::**bessel_Yn_e** x
implements gsl_sf_bessel_Yn_e.

gsl::sf::**bessel_Yn_array** nmin::int nmax::int x
implements gsl_sf_bessel_Yn_array.

gsl::sf::**bessel_I0** x
implements gsl_sf_bessel_I0.

gsl::sf::**bessel_I0_e** x
implements gsl_sf_bessel_I0_e.

gsl::sf::**bessel_I1** x
implements gsl_sf_bessel_I1.

gsl::sf::**bessel_I1_e** x
implements gsl_sf_bessel_I1_e.

gsl::sf::**bessel_In** n::int x
implements gsl_sf_bessel_In.

gsl::sf::**bessel_In_e** n::int x
implements gsl_sf_bessel_In_e

`gsl::sf::`**`bessel_In_array`** nmin::int nmax::int x
implements `gsl_sf_bessel_In_array`.

`gsl::sf::`**`bessel_I0_scaled`** x
implements `gsl_sf_bessel_I0_scaled`.

`gsl::sf::`**`bessel_I0_scaled_e`** x
implements `gsl_sf_bessel_I0_scaled_e`.

`gsl::sf::`**`bessel_I1_scaled`** x
implements `gsl_sf_bessel_I1_scaled`.

`gsl::sf::`**`bessel_I1_scaled_e`** x
implements `gsl_sf_bessel_I1_scaled_e`.

`gsl::sf::`**`bessel_In_scaled`** n::int x
implements `gsl_sf_bessel_In_scaled`.

`gsl::sf::`**`bessel_In_scaled_e`** n::int x
implements `gsl_sf_bessel_In_scaled_e`.

`gsl::sf::`**`bessel_In_scaled_array`** nmin::int nmax::int x
implements `gsl_sf_bessel_In_array`.

`gsl::sf::`**`bessel_K0`** x
implements `gsl_sf_bessel_K0`.

`gsl::sf::`**`bessel_K0_e`** x
implements `gsl_sf_bessel_K0_e`.

`gsl::sf::`**`bessel_K1`** x
implements `gsl_sf_bessel_K1`.

`gsl::sf::`**`bessel_K1_e`** x
implements `gsl_sf_bessel_K1_e`.

`gsl::sf::`**`bessel_Kn`** n::int x
implements `gsl_sf_bessel_Kn`.

`gsl::sf::`**`bessel_Kn_e`** n::int x
implements `gsl_sf_bessel_Kn_e`

`gsl::sf::`**`bessel_Kn_array`** nmin::int nmax::int x
implements `gsl_sf_bessel_Kn_array`.

`gsl::sf::`**`bessel_K0_scaled`** x
implements `gsl_sf_bessel_K0_scaled`.

`gsl::sf::`**`bessel_K0_scaled_e`** x
implements `gsl_sf_bessel_K0_scaled_e`.

`gsl::sf::`**`bessel_K1_scaled`** x
implements `gsl_sf_bessel_K1_scaled`.

`gsl::sf::`**`bessel_K1_scaled_e`** x
implements `gsl_sf_bessel_K1_scaled_e`.

`gsl::sf::`**`bessel_Kn_scaled`** n::int x
implements `gsl_sf_bessel_Kn_scaled`.

`gsl::sf::`**`bessel_Kn_scaled_e`** n::int x
implements `gsl_sf_bessel_Kn_scaled_e`.

`gsl::sf::`**`bessel_Kn_scaled_array`** nmin::int nmax::int x
implements `gsl_sf_bessel_Kn_array`.

`gsl::sf::`**`bessel_j0`** x
implements `gsl_sf_bessel_j0`.

`gsl::sf::`**`bessel_j0_e`** x
implements `gsl_sf_bessel_j0_e`.

`gsl::sf::`**`bessel_j1`** x
implements `gsl_sf_bessel_j1`.

`gsl::sf::`**`bessel_j1_e`** x
implements `gsl_sf_bessel_j1_e`.

`gsl::sf::`**`bessel_j2`** x
implements `gsl_sf_bessel_j2`.

`gsl::sf::`**`bessel_j2_e`** x
implements `gsl_sf_bessel_j2_e`.

`gsl::sf::`**`bessel_jl`** l::int x
implements `gsl_sf_bessel_jl`.

`gsl::sf::`**`bessel_jl_e`** l::int x
implements `gsl_sf_bessel_jl_e`.

`gsl::sf::`**`bessel_jl_array`** lmax::int x
implements `gsl_sf_bessel_jl_array`.

`gsl::sf::`**`bessel_jl_steed_array`** lmax::int x
implements `gsl_sf_bessel_jl_steed_array`.

`gsl::sf::`**`bessel_y0`** x
implements `gsl_sf_bessel_y0`.

`gsl::sf::`**`bessel_y0_e`** x
implements `gsl_sf_bessel_y0_e`.

`gsl::sf::`**`bessel_y1`** x
implements `gsl_sf_bessel_y1`.

`gsl::sf::`**`bessel_y1_e`** x
implements `gsl_sf_bessel_y1_e`.

`gsl::sf::`**`bessel_y2`** x
implements `gsl_sf_bessel_y2`.

`gsl::sf::`**`bessel_y2_e`** x
implements `gsl_sf_bessel_y2_e`.

`gsl::sf::`**`bessel_yl`** l::int x
: implements `gsl_sf_bessel_yl`.

`gsl::sf::`**`bessel_yl_e`** l::int x
: implements `gsl_sf_bessel_yl_e`.

`gsl::sf::`**`bessel_yl_array`** lmax::int x
: implements `gsl_sf_bessel_yl_array`.

`gsl::sf::`**`bessel_i0_scaled`** x
: implements `gsl_sf_bessel_i0_scaled`.

`gsl::sf::`**`bessel_i0_scaled_e`** x
: implements `gsl_sf_bessel_i0_scaled_e`.

`gsl::sf::`**`bessel_i1_scaled`** x
: implements `gsl_sf_bessel_i1_scaled`.

`gsl::sf::`**`bessel_i1_scaled_e`** x
: implements `gsl_sf_bessel_i1_scaled_e`.

`gsl::sf::`**`bessel_i2_scaled`** x
: implements `gsl_sf_bessel_i2_scaled`.

`gsl::sf::`**`bessel_i2_scaled_e`** x
: implements `gsl_sf_bessel_i2_scaled_e`.

`gsl::sf::`**`bessel_il_scaled`** l::int x
: implements `gsl_sf_bessel_il_scaled`.

`gsl::sf::`**`bessel_il_scaled_e`** l::int x
: implements `gsl_sf_bessel_il_scaled_e`.

`gsl::sf::`**`bessel_il_scaled_array`** lmax::int x
: implements `gsl_sf_bessel_il_scaled_array`.

`gsl::sf::`**`bessel_k0_scaled`** x
: implements `gsl_sf_bessel_k0_scaled`.

`gsl::sf::`**`bessel_k0_scaled_e`** x
: implements `gsl_sf_bessel_k0_scaled_e`.

`gsl::sf::`**`bessel_k1_scaled`** x
: implements `gsl_sf_bessel_k1_scaled`.

`gsl::sf::`**`bessel_k1_scaled_e`** x
: implements `gsl_sf_bessel_ik_scaled_e`.

`gsl::sf::`**`bessel_k2_scaled`** x
: implements `gsl_sf_bessel_k2_scaled`.

`gsl::sf::`**`bessel_k2_scaled_e`** x
: implements `gsl_sf_bessel_k2_scaled_e`.

`gsl::sf::`**`bessel_kl_scaled`** l::int x
: implements `gsl_sf_bessel_kl_scaled`.

`gsl::sf::`**`bessel_kl_scaled_e`** l::int x
implements `gsl_sf_bessel_kl_scaled_e`.

`gsl::sf::`**`bessel_kl_scaled_array`** lmax::int x
implements `gsl_sf_bessel_il_scaled_array`.

`gsl::sf::`**`bessel_Jnu`** nu x
implements `gsl_sf_bessel_Jnu`.

`gsl::sf::`**`bessel_Jnu_e`** nu x
implements `gsl_sf_bessel_Jnu_e`.

`gsl::sf::`**`bessel_sequence_Jnu_e`** nu v::matrix
implements `gsl_sf_bessel_sequence_Jnu_e`.

`gsl::sf::`**`bessel_Ynu`** nu x
implements `gsl_sf_bessel_Ynu`.

`gsl::sf::`**`bessel_Ynu_e`** nu x
implements `gsl_sf_bessel_Ynu_e`.

`gsl::sf::`**`bessel_Inu`** nu x
implements `gsl_sf_bessel_Inu`.

`gsl::sf::`**`bessel_Inu_e`** nu x
implements `gsl_sf_bessel_Inu_e`.

`gsl::sf::`**`bessel_Inu_scaled`** nu x
implements `gsl_sf_bessel_Inu_scaled`.

`gsl::sf::`**`bessel_Inu_scaled_e`** nu x
implements `gsl_sf_bessel_Inu_scaled_e`.

`gsl::sf::`**`bessel_Knu`** nu x
implements `gsl_sf_bessel_Knu`.

`gsl::sf::`**`bessel_Knu_e`** nu x
implements `gsl_sf_bessel_Knu`.

`gsl::sf::`**`bessel_lnKnu`** nu x
implements `gsl_sf_bessel_lnKnu`.

`gsl::sf::`**`bessel_lnKnu_e`** nu x
implements `gsl_sf_bessel_lnKnu_e`.

`gsl::sf::`**`bessel_Knu_scaled`** nu x
implements `gsl_sf_bessel_Knu_scaled`.

`gsl::sf::`**`bessel_Knu_scaled_e`** nu x
implements `gsl_sf_bessel_Knu_scaled_e`.

`gsl::sf::`**`bessel_zero_J0`** s::int
implements `gsl_sf_bessel_zero_J0`.

`gsl::sf::`**`bessel_zero_J0_e`** s::int
implements `gsl_sf_bessel_zero_J0_e`.

gsl::sf::**bessel_zero_J1** s::int
: implements gsl_sf_bessel_zero_J1.

gsl::sf::**bessel_zero_J1_e** s::int
: implements gsl_sf_bessel_zero_J1_e.

gsl::sf::**bessel_zero_Jnu** nu s::int
: implements gsl_sf_bessel_zero_Jnu.

gsl::sf::**bessel_zero_Jnu_e** nu s::int
: implements gsl_sf_bessel_zero_Jnu.

15.2.4 Examples

The following illustrate the Bessel functions.

```
> using gsl::sf;
> using namespace gsl::sf;
> bessel_J0 (-1.2);
0.671132744264363
> bessel_J0_e 0.75;
[0.864242275166649,7.07329111491049e-16]
> bessel_J1 1.2;
0.498289057567216
> bessel_J1_e (-0.2);
[-0.099500832639236,5.00768737808415e-17]
> bessel_Jn 0 (-1.2);
0.671132744264363
> bessel_Jn_e 2 0.75;
[0.0670739972996506,5.48959386474892e-17]
> bessel_Jn_array 0 4 0.5;
[0.938469807240813,0.242268457674874,0.0306040234586826,
 0.00256372999458724,0.000160736476364288]
> bessel_Y0 0.25;
-0.931573024930059
> bessel_Y0_e 0.25;
[-0.931573024930059,6.4279898430593e-16]
> bessel_Y1 0.125;
-5.19993611253477
> bessel_Y1_e 4.325;
[0.343041276811844,2.74577716760089e-16]
> bessel_Yn 3 4.325;
-0.0684784962694202
> bessel_Yn_e 3 4.325;
[-0.0684784962694202,3.37764590906247e-16]
> bessel_Yn_array 2 4 1.35;
[-1.07379345815726,-2.66813016175689,-10.7845628163178]
> bessel_I0 1.35;
1.51022709775726
> bessel_I0_e 1.35;
[1.51022709775726,2.37852166449918e-15]
> bessel_I1 0.35;
```

```
0.177693400031422
> bessel_I1_e 0.35;
[0.177693400031422,1.55520651386126e-16]
> bessel_In 2 3.0;
2.24521244092995
> bessel_In_e 2 3.0;
2.24521244092995,5.98244771302867e-15]
> bessel_In_array 3 5 (-0.1);
[-2.08463574223272e-05,2.60546902129966e-07,-2.6052519298937e-09]
> bessel_I0_scaled 1.05;
0.453242541279856
> bessel_I0_scaled_e 1.05;
[0.453242541279856,4.10118141697477e-16]
> bessel_I1_scaled 1.05;
0.210226017612868
> bessel_I1_scaled_e 1.05;
[0.210226017612868,2.12903131803686e-16]
> bessel_In_scaled 3 1.05;
0.00903732602788281
> bessel_In_scaled_e 3 1.05;
[0.00903732602788281,2.00668948743994e-17]
> bessel_In_scaled_array 3 5 1.05;
[0.00903732602788281,0.0011701685245855,0.000121756316755217]
> bessel_K0 2.3;
0.0791399330020936
> bessel_K0_e 2.3;
[0.0791399330020936,1.15144454318261e-16]
> bessel_K1 2.3;
0.0949824438453627
> bessel_K1_e 2.3;
[0.0949824438453627,9.85583638959967e-17]
> bessel_Kn 2 3.4;
0.0366633035851529
> bessel_Kn_e 2 3.4;
[0.0366633035851529,2.01761856558251e-16]
> bessel_Kn_array 1 3 2.5;
[0.0738908163477471,0.121460206278564,0.268227146393449]
> bessel_K0_scaled 1.5;
0.367433609054158
> bessel_K0_scaled_e 1.5;
[0.958210053294896,1.25816573186951e-14]
> bessel_K1_scaled 1.5;
1.24316587355255
> bessel_K1_scaled_e 1.5;
[1.24316587355255,2.32370553362606e-15]
> bessel_Kn_scaled 4 1.5;
35.4899165934682
> bessel_Kn_scaled_e 4 1.5;
[35.4899165934682,3.89252285021454e-14]
> bessel_Kn_scaled_array 4 6 1.5;
[35.4899165934682,197.498093175689,1352.14387109806]
> bessel_j0 0.01;
```

```
0.999983333416666
> bessel_j0_e 0.01;
[0.999983333416666,4.44081808400239e-16]
> bessel_j1 0.2;
0.0664003806703222
> bessel_j1_e 0.2;
[0.0664003806703222,2.94876925856268e-17]
> bessel_j2 0.3;
0.00596152486862022
> bessel_j2_e 0.3;
[0.00596152486862022,2.64744886840705e-18]
> bessel_jl 4 0.3;
8.53642426502516e-06
> bessel_jl_e 4 0.3;
[8.53642426502516e-06,1.02355215483598e-19]
> bessel_jl_array 2 1.2;
[0.776699238306022,0.34528456985779,0.0865121863384538]
> bessel_jl_steed_array 2 1.2;
[0.776699238306022,0.34528456985779,0.0865121863384538]
> bessel_y0 1;
-0.54030230586814
> bessel_y0_e 3;
[0.329997498866815,2.93096657048522e-16]
> bessel_y1 3;
0.062959163602316
> bessel_y1_e 3.0;
[0.062959163602316,1.04609100698801e-16]
> bessel_yl 3 5;
-0.0154429099129942
> bessel_yl_e 3 5;
[-0.0154429099129942,2.87258769784673e-17]
> bessel_i0_scaled 3;
0.166253541303889
> bessel_i0_scaled_e 3;
[0.166253541303889,7.38314037924188e-17]
> bessel_i1_scaled 3;
0.111661944928148
> bessel_i1_scaled_e 3;
[0.111661944928148,4.95878648934625e-17]
> bessel_i2_scaled 3;
0.0545915963757409
> bessel_i2_scaled_e 3;
[0.0545915963757409,2.42435388989563e-17]
> bessel_il_scaled 3 1;
0.0037027398773348
> bessel_il_scaled_e 3 1;
[0.0037027398773348,8.46838615599053e-17]
> bessel_il_scaled_array 3 1;
[0.432332358381693,0.135335283236613,0.0263265086718556,0.0037027398773348]
> bessel_k0_scaled 3;
0.523598775598299
> bessel_k0_scaled_e 3;
```

```
[0.523598775598299,2.32524566533909e-16]
> bessel_k1_scaled 4;
0.490873852123405
> bessel_k1_scaled_e 4;
[0.490873852123405,2.17991781125539e-16]
> bessel_k2_scaled 4;
0.760854470791278
> bessel_k2_scaled_e 4;
[0.760854470791278,3.37887260744586e-16]
> bessel_kl_scaled 2 4;
0.760854470791278
> bessel_kl_scaled_e 2 4;
[0.760854470791278,3.37887260744586e-16]
> bessel_kl_scaled_array 2 4;
[0.392699081698724,0.490873852123405,0.760854470791278]
> bessel_Jnu 2 2.3;
0.413914591732062
> bessel_Jnu_e 2 2.3;
[0.413914591732062,6.43352513956959e-16]
> bessel_sequence_Jnu_e 2 {.1,.2,.3};
[0.00124895865879992,0.00498335415278356,0.011165861949064]
> bessel_Ynu 1 0.5;
-1.47147239267024
> bessel_Ynu_e 1 0.5;
[-1.47147239267024,8.49504515830242e-15]
> bessel_Inu 1.2 3.4;
5.25626563437082
> bessel_Inu_e 1.2 3.4;
[5.25626563437082,1.00839636820646e-13]
> bessel_Inu_scaled 1.2 3.4;
0.175418771999042
> bessel_Inu_scaled_e 1.2 3.4;
[0.175418771999042,3.15501414592188e-15]
> bessel_Knu 3 3;
0.122170375757184
> bessel_Knu_e 3 3;
[0.122170375757184,4.34036365096743e-16]
> bessel_lnKnu 3 3;
-2.10233868587978
> bessel_lnKnu_e 3 3;
[-2.10233868587978,4.24157124665032e-15]
> bessel_Knu_scaled 3 3;
2.45385759319062
> bessel_Knu_scaled_e 3 3;
[2.45385759319062,7.6281217575122e-15]
> bessel_zero_J0 3;
8.65372791291102
> bessel_zero_J0_e 3;
[8.65372791291102,2.59611837387331e-14]
> bessel_zero_J1 3;
10.1734681350627
> bessel_zero_J1_e 3;
```

```
[10.1734681350627,2.03469362701254e-13]
> bessel_zero_Jnu 1.2 3;
10.46769
> bessel_zero_Jnu_e 1.2 3;
[10.4676986203553,2.09353972407105e-14]86203553
```

15.2.5 Clausen Functions

`gsl::sf::`**`clausen`** x
: implements `gsl_sf_clausen`.

`gsl::sf::`**`clausen_e`** x
: implements `gsl_sf_clausen_e`.

15.2.6 Examples

The following illustrate the Clausen functions.

```
> using gsl::sf;
> using namespace gsl::sf;
> clausen 4.5;
-0.831839220823219
> clausen_e 4.5;
[-0.831839220823219,8.60688668835964e-16]
```

15.2.7 Colomb Functions

The results of the Coulomb wave functions are returned as a list whose elements are ordered corresponding to the argument order of the corresponding C functions in GSL library.

`gsl::sf::`**`hydrogenicR_1`** Z r
: implements `gsl_sf_hydrogenicR_1`.

`gsl::sf::`**`hydrogenicR_1_e`** Z r
: implements `gsl_sf_hydrogenicR_1_e`.

`gsl::sf::`**`hydrogenicR`** n::int l::int Z r
: implements `gsl_sf_hydrogenicR_1`.

`gsl::sf::`**`hydrogenicR_e`** n::int l::int Z r
: implements `gsl_sf_hydrogenicR_1_e`.

`gsl::sf::`**`coulomb_wave_FG_e`** eta x L_F k::int
: implements `gsl_sf_coulomb_wave_FG_e`.

`gsl::sf::`**`coulomb_wave_F_array`** L_min kmax::int eta x
: implements `gsl_sf_coulomb_wave_F_array`.

`gsl::sf::`**`coulomb_wave_FG_array`** L_min kmax::int eta x
: implements `gsl_sf_coulomb_wave_FG_array`.

`gsl::sf::`**`coulomb_wave_FGp_array`** L_min kmax::int eta x
: implements `gsl_sf_coulomb_wave_FGp_array`.

`gsl::sf::`**`coulomb_wave_sphF_array`** L_min kmax::int eta x
: implements `gsl_sf_coulomb_wave_sphF_array`.

`gsl::sf::`**`coulomb_CL_e`** L eta
: implements `gsl_sf_coulomb_wave_CL_e`.

`gsl::sf::`**`coulomb_CL_array`** Lmin kmax eta
: implements `gsl_sf_coulomb_wave_CL_array`.

15.2.8 Examples

The following illustrate the Coulomb functions.

```
> using gsl::sf;
> using namespace gsl::sf;
> hydrogenicR_1 0.2 4;
0.0803784086420537
> hydrogenicR_1_e 0.2 4;
[0.0803784086420537,2.85561471862841e-17]
> hydrogenicR 3 1 0.25 3.2;
0.00802954301593587
> hydrogenicR_e 3 1 0.25 3.2;
[0.00802954301593587,3.90138748076797e-17]
> coulomb_wave_F_array 1 2 0.5 0.5;
[{0.0387503306520188,0.0038612830533923,0.000274978904710252},0.0]
> coulomb_wave_FG_array 1 2 0.5 0.5;
[{0.0387503306520188,0.0038612830533923,0.000274978904710252},
 {4.13731494044202,25.4479852847406,257.269816591168},0.0,0.0]
> coulomb_wave_FGp_array 1 2 0.5 0.5;
[{0.0387503306520188,0.0038612830533923,0.000274978904710252},
 {4.13731494044202,25.4479852847406,257.269816591168},0.0,0.0]
> coulomb_wave_sphF_array 1 2 0.5 0.5;
[{0.0775006613040376,0.0077225661067846,0.000549957809420504},0.0]
> coulomb_CL_e (-0.5) 3;
[0.000143036170217949,2.92195771135514e-18]
> coulomb_CL_array (-0.5) 4 1.5;
[0.0159218263353144,0.0251746178646226,0.00890057150292734,
 0.00172996014234001,0.000235267570111599]
```

15.2.9 Coupling Coefficients

`gsl::sf::`**`coupling_3j`** m::matrix
: implements `gsl_sf_coupling_3j` except the input is a 2x3 (row by column) integer matrix instead of six integer arguments.

`gsl::sf::`**`coupling_3j_e`** m::matrix
: implements `gsl_sf_coupling_3j_e` except the input is a 2x3 (row by column) integer matrix instead of six integer arguments.

`gsl::sf::`**`coupling_6j`** m::matrix
: implements `gsl_sf_coupling_6j` except the input is a 2x3 (row by column) integer matrix instead of six integer arguments.

`gsl::sf::`**`coupling_6j_e`** m::matrix
: implements `gsl_sf_coupling_6j_e` except the input is a 2x3 (row by column) integer matrix instead of six integer arguments.

`gsl::sf::`**`coupling_9j`** m::matrix
: implements `gsl_sf_coupling_9j` except the input is a 3x3 integer matrix instead of six integer arguments.

`gsl::sf::`**`coupling_9j_e`** m::matrix
: implements `gsl_sf_coupling_9j_e` except the input is a 3x3 integer matrix instead of six integer arguments.

15.2.10 Examples

The following illustrate the coupling coefficient functions.

```
> using gsl::sf;
> using namespace gsl::sf;
> coupling_3j {6,4,2;0,0,0};
-0.29277002188456
> coupling_3j_e {6,4,2;0,0,0};
[-0.29277002188456,1.300160076865e-16]
> coupling_6j {1,2,3;2,1,2};
-0.166666666666667
> coupling_6j_e {1,2,3;2,1,2};
[-0.166666666666667,2.22044604925031e-16]
> coupling_9j {1,2,3;2,1,2;1,1,1};
-0.0962250448649376
> coupling_9j_e {1,2,3;2,1,2;1,1,1};
[-0.0962250448649376,4.84948508304183e-16]
```

15.2.11 Dawson Function

`gsl::sf::`**`dawson`** x
: implements `gsl_sf_dawson`.

`gsl::sf::`**`dawson_e`** x
: implements `gsl_sf_dawson_e`.

15.2.12 Examples

The following illustrate the dawson functions.

```
> dawson 3;/**-
0.178271030610558
> dawson_e 3;
[0.178271030610558,8.9920386788099e-16]
```

15.2.13 Debye Functions

`gsl::sf::`**debye_1** x
: implements `gsl_sf_debye_1`.

`gsl::sf::`**debye_1_e** x
: implements `gsl_sf_debye_1_e`.

`gsl::sf::`**debye_2** x
: implements `gsl_sf_debye_2`.

`gsl::sf::`**debye_2_e** x
: implements `gsl_sf_debye_2_e`.

`gsl::sf::`**debye_3** x
: implements `gsl_sf_debye_3`.

`gsl::sf::`**debye_3_e** x
: implements `gsl_sf_debye_3_e`.

`gsl::sf::`**debye_4** x
: implements `gsl_sf_debye_4`.

`gsl::sf::`**debye_4_e** x
: implements `gsl_sf_debye_4_e`.

`gsl::sf::`**debye_5** x
: implements `gsl_sf_debye_5`.

`gsl::sf::`**debye_5_e** x
: implements `gsl_sf_debye_5_e`.

`gsl::sf::`**debye_6** x
: implements `gsl_sf_debye_6`.

`gsl::sf::`**debye_6_e** x
: implements `gsl_sf_debye_6_e`.

15.2.14 Examples

The following illustrate the debye functions.

```
> debye_1 0.4;
0.904437352623294
> debye_1_e 0.4;
[0.904437352623294,3.84040456356756e-16]
> debye_2 1.4;
0.613281386045505
> debye_2_e 1.4;
[0.613281386045505,5.15090106564116e-16]
> debye_3 2.4;
0.370136882985216
> debye_3_e 2.4;
[0.370136882985216,6.0792125556598e-16]
> debye_4 3.4;
0.205914922541978
> debye_4_e 3.4;
[0.205914922541978,7.42872979584512e-16]
> debye_5 4.4;
0.107477287722471
> debye_5_e 4.4;
[0.107477287722471,2.38647518907499e-17]
> debye_6 5.4;
0.0533132925698824
> debye_6_e 5.4;
[0.0533132925698824,1.18379289859322e-17]
```

15.2.15 Dilogarithm

`gsl::sf::`**`dilog`** x
: implements `gsl_sf_dilog`.

`gsl::sf::`**`dilog`** (r<:theta)
: implements `gsl_sf_complex_dilog_e` except that results are returned as the complex value `re+:im` and the error values are not returned.

`gsl::sf::`**`dilog_e`** x
: implements `gsl_sf_dilog_e`.

`gsl::sf::`**`dilog_e`** (r<:theta)
: implements `gsl_sf_complex_dilog_e` except the results are returned as the list `[re+:im, re_error, im_error]`.

15.2.16 Examples

The following illustrate the dilog functions.

```
> dilog 1.0;
1.64493406684823
> dilog (1<:2);
-0.496658586741567+:0.727146050863279
```

```
> dilog_e (1%3);
[0.366213229977064,8.22687466397711e-15]
> dilog_e (1<:3);
[-0.817454913536463+:0.0980262093913011,3.8224192909699e-15,
 1.47247478976757e-15]
```

`gsl::sf::`**`multiply_e`** x y
: implements `gsl_sf_multiply_e`.

`gsl::sf::`**`multiply_err_e`** x dx y dy
: implements `gsl_sf_multiply_err_e`.

15.2.17 Examples

The following illustrate the multiply functions.

```
> multiply_e 10.0 11.0;
[110.0,4.88498130835069e-14]
> multiply_err_e 10.0 0.04 11.0 0.002;
[110.0,0.460000000000049]
```

15.3 Matrices

This module is loaded via the command `using gsl::matrix` and provides wrappers for many of the GSL matrix, BLAS, and linear algebra routines found in Chapters 8, 12, and 13, respectively of the GSL Reference Manual:

- Vectors and Matrices
- BLAS Support
- Linear Algebra

It also contains some general utility functions for creating various types of matrices.

15.3.1 Matrix Creation

The utility functions `zeros` and `ones` create matrices with all elements zero or one, respectively, and `eye` creates identity matrices. These functions can be invoked either with a pair **(n,m)** denoting the desired number of rows or columns, or an integer **n** in which case a square **n** x **n** matrix is created. The result is always a double matrix. Analogous functions `izeros`, `czeros`, etc. are provided to create integer and complex matrices, respectively.

`gsl::matrix::`**`zeros`** (n :: int, m :: int)
: creates an **n** x **m** double matrix with all of its entries being zero.

`gsl::matrix::`**`zeros`** n :: int
: creates an **n** x **n** double matrix with all of its entries being zero.

`gsl::matrix::`**`izeros`** (n :: int, m :: int)
creates an **n** x **m** integer matrix with all of its entries being zero.

`gsl::matrix::`**`izeros`** n :: int
creates an **n'x'n** integer matrix with all of its entries being zero.

`gsl::matrix::`**`czeros`** (n :: int, m :: int)
creates an **n** x **m** complex matrix with all of its entries being zero.

`gsl::matrix::`**`czeros`** n :: int
creates an **n** x **n** complex matrix with all of its entries being zero.

`gsl::matrix::`**`ones`** (n :: int, m :: int)
creates an **n** x **m** double matrix with all of its entries being one.

`gsl::matrix::`**`ones`** n :: int
creates an **n** x **n** double matrix with all of its entries being one.

`gsl::matrix::`**`iones`** (n :: int, m :: int)
creates an **n** x **m** integer matrix with all of its entries being one.

`gsl::matrix::`**`iones`** n :: int
creates an **n** x **n** integer matrix with all of its entries being one.

`gsl::matrix::`**`cones`** (n :: int, m :: int)
creates an **n** x **m** complex matrix with all of its entries being one.

`gsl::matrix::`**`cones`** n :: int
creates an **n** x **n** complex matrix with all of its entries being one.

`gsl::matrix::`**`eye`** (n :: int, m :: int)
creates an **n** x **m** identity matrix with double entries.

`gsl::matrix::`**`eye`** n :: int
creates an **n** x **n** identity matrix with double entries.

`gsl::matrix::`**`ieye`** (n :: int, m :: int)
creates an **n** x **m** identity matrix with integer entries.

`gsl::matrix::`**`ieye`** n :: int
creates an **n** x **n** identity matrix with integer entries.

`gsl::matrix::`**`ceye`** (n :: int, m :: int)
creates an **n** x **m** identity matrix with complex entries.

`gsl::matrix::`**`ceye`** n :: int
creates an **n** x **n** identity matrix with complex entries.

15.3.2 Matrix Operators and Functions

The following operations are defined for constant `a` and matrices `x` and `y`. Some operators are not defined in the GSL library but are provided here for convenience.

a **+** x

x **+** a
: returns a matrix with entries `a + x!(i,j)`.

x **+** y
: adds matrix `x` to matrix `y`.

- x
: returns a matrix with entries `- x!(i,j)`. Note that `neg x` is equivalent to `- x`.

a **-** x
: returns a matrix with entries `a - x!(i,j)`.

x **-** a
: returns a matrix with entries `x!(i,j) - a`.

x **-** y
: subtracts matrix `y` from matrix `x`.

a ∗ x
x ∗ a
: returns a matrix with entries `a ∗ x!(i,j)`.

x **.*** y
: multiplies, element-wise, matrix `x` to matrix `y`.

x ∗ y
: multiplies matrix `x` to matrix `y`.

a **/** x
: returns a matrix with entries `a / x!(i,j)`. Note that matrix `x` must not have any zero entries.

x **/** a
: returns a matrix with entries `x!(i,j) / a`. Note that `a` must be nonzero.

x **./** y
: divides, element-wise, matrix **x** by matrix **y**.

x **/** y
: right divides matrix **x** by matrix **y**.

x **** y
: left divides matrix **x** by matrix **y**.

a **div** x
: returns an integer matrix with entries `a div x!(i,j)`. Note that **a** must be an integer and matrix `x` must be an integer matrix with nonzero entries.

x **div** a
: returns an integer matrix with entries `x!(i,j) div a`. Note that `a` must be a nonzero integer and matrix `x` must have integer entries.

x **div** y
: computes the quotient integer matrix `x` by integer matrix `y`.

a **mod** x
: returns an integer matrix with entries `a mod x!(i,j)`. Note that `a` must be an integer and matrix `x` must be an integer matrix with nonzero entries.

x **mod** a
: returns an integer matrix with entries `a mod x!(i,j)`. Note that `a` must be an integer and matrix `x` must be an integer matrix with nonzero entries.

x **mod** y
: returns the remainder integer matrix `x` mod integer matrix `y`.

not x
: returns a matrix with integer entries `not x!(i,j)`. Note that `x` must be a matrix with integer entries and `not` is the bitwise negation operation.

a **^** x
: returns a matrix with entries `a ^ x!(i,j)`. Note that `0^0` is defined as 1.

x **^** a
: returns a matrix with entries `x!(i,j) ^ a`. Note that `0^0` is defined as 1.

x **.^** y
: returns a matrix with entries `x!(i,j) ^ y!(i,j)`.

x **^** y
: returns a matrix with entries `x!(i,j) ^ y!(i,j)`.

x **<<** a
: returns an integer matrix with entries `x!(i,j) << a`. Note that `a` must be an integer and matrix `x` must have integer entries.

x **<<** y
: returns an integer matrix with entries `x!(i,j) << y!(i,j)`. Note that `x` and `y` must have integer entries.

x **>>** a
: returns an integer matrix with entries `x!(i,j) >> a`. Note that `a` must be an integer and matrix `x` must have integer entries.

x **>>** y
: returns an integer matrix with entries `x!(i,j) >> y!(i,j)`. Note that `x` and `y` must have integer entries.

x **and** a
a **and** x
: returns an integer matrix with entries `a and x!(i,j)`. Note that `a` must be an integer, matrix `x` must have integer entries, and `and` is a bitwise operator.

x **and** y
: returns an integer matrix with entries `x!(i,j) and y!(i,j)`. Note that `x` and `y` must be matrices with integer entries.

x **or** a

a **or** x
: returns an integer matrix with entries `a or x!(i,j)`. Note that `a` must be an integer, matrix `x` must have integer entries, and `or` is a bitwise operator.

x **or** y
: returns an integer matrix with entries `x!(i,j) or y!(i,j)`. Note that `x` and `y` must be matrices with integer entries.

The `pow` function computes powers of matrices by repeated matrix multiplication.

pow x :: matrix k :: int
pow x :: matrix k :: bigint
: Raises matrix `x` to the `k` th power. Note `x` must be a square matrix and `k` a nonnegative integer.

15.3.3 Singular Value Decomposition

For a given **n** x **m** matrix `x`, these functions yield a singular-value decomposition `u`, `s`, `v` of the matrix such that `x == u*s*transpose v`, where `u` and `v` are orthogonal matrices of dimensions **n** x **m** and **n** x **n**, respectively, and **s** is a **n** x **n** diagonal matrix which has the singular values in its diagonal, in descending order. Note that GSL implements this only for double matrices right now. Also, GSL only handles the case of square or overdetermined systems, but we work around that in our wrapper functions by just adding a suitable number of zero rows in the underdetermined case.

`gsl::matrix::`**svd** x
: singular-value decomposition of matrix `x`.

`gsl::matrix::`**svd_mod** x
: This uses the modified Golub-Reinsch algorithm, which is faster if `n > m` but needs **O(m^2)** extra memory as internal workspace.

`gsl::matrix::`**svd_jacobi** x
: This uses one-sided Jacobi orthogonalization which provides better relative accuracy but is slower.

`gsl::matrix::`**svd_solve** (u, s, v) b
: Solve the system `Ax=b`, using the SVD of `A`. `svd_solve` takes the result `(u,s,v)` of a `svd` call, and a column vector `b` of the appropriate dimension. The result is another column vector solving the system (possibly in the least-squares sense).

`gsl::matrix::`**pinv** x
: Computes the pseudo inverse of a matrix from its singular value decomposition.

15.4 Least-Squares Fitting

This module is loaded via the command `using gsl::fit` and provides Pure wrappers for the GSL least-squares fitting routines found in Chapter 36 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Least_002dSquares-Fitting.html.

15.4.1 Routines

`gsl::fit::`**`linear`** x::matrix y::matrix
: implements `gsl_fit_linear` without the `xstride`, `ystride`, and `n` parameters. Results are returned as a list `[c0, c1, cov00, cov01, cov11, sumsq]`.

`gsl::fit::`**`wlinear`** x::matrix w::matrix y::matrix
: implements `gsl_fit_wlinear` without the `xstride`, `wstride`, `ystride`, and `n` parameters. Results are given as a list `[c0, c1, cov00, cov01, cov11, chisq]`.

`gsl::fit::`**`linear_est`** x c0::double c1::double cov00::double cov01::double cov11::double
: implements `gsl_fit_linear_est`. Results are returned as a list `[y, y_err]`.

`gsl::fit::`**`mul`** x::matrix y::matrix
: implements `gsl_fit_mul` omitting the parameters `xstride`, `ystride`, and `n`. Results are returned as a list `[c1, cov11, sumsq]`.

`gsl::fit::`**`wmul`** x::matrix w::matrix y::matrix
: implements `gsl_fit_wmul` omitting the parametrs `xstride`, `ystride`, and `n`. Results are returned as a list `[c1, cov11, sumsq]`.

`gsl::fit::`**`mul_est`** x c1::double cov11::double
: implements `gsl_fit_mul_est`. Results are returned as a list `[y, y_err]`.

15.4.2 Examples

Usage of each implemented library routine is illustrated below.

```
> using gsl::fit;
> using namespace gsl::fit;
```

The following code determines the equation for the least-squares line through the points (1,0.01), (2,1.11), (3,1.9), (4,2.85), and (5,4.01).

```
> Y x = '(a + b * x)
> when
>   a:b:_ = linear {1,2,3,4,5} {0.01,1.11,1.9,2.85,4.01}
> end;
> Y x;
-0.946+0.974*x
> eval $ Y 2;
1.002
```

The following code illustrates estimating y-values without constructing an equation for the least-squares line determined by the points `{x1,x2,x3,...,xn}`, `{y1,y2,y3,...,yn}`. Here we estimate the **y**-value at **x** = 1, **x** = 2, and **x** = 3. Compare the output above at **x** = 2 to the output at **x** = 2 below.

```
> let c0:c1:cov00:cov01:cov11:_ = linear {1,2,3,4,5}
>   {0.01,1.11,1.9,2.85,4.01};
> linear_est 1 c0 c1 cov00 cov01 cov11;
[0.028,0.0838570211729465]
> linear_est 2 c0 c1 cov00 cov01 cov11;
[1.002,0.0592958683214944]
> linear_est 3 c0 c1 cov00 cov01 cov11;
[1.976,0.0484148737476408]
```

Next, we determine a least-squares line through the points (1,0.01), (2,1.11), (3,1.9), (4,2.85), and (5,4.01) using weights 0.1, 0.2, 0.3, 0.4, and 0.5.

```
> W x = '(a + b * x)
> when
>   a:b:_ = wlinear (matrix (1..5))
>           {0.1, 0.2, 0.3, 0.4, 0.5}
>           {0.01, 1.11, 1.9, 2.85, 4.01};
> end;
> W u;
-0.99+0.986*u
> eval $ W 2;
0.982
```

The least-squares slope for `Y = c1 * X` using the points (1,3), (2,5), and (3,7) is calculated below. Also, the **y**-values and standard error about **x** = 1, 2, and 3 are given.

```
> let c1:cov11:sumsq:_ = mul {1,2,3} {3,5,7};
> mul_est 1 c1 cov11;
[2.42857142857143,0.123717914826348]
> mul_est 2 c1 cov11;
[4.85714285714286,0.247435829652697]
> mul_est 3 c1 cov11;
[7.28571428571428,0.371153744479045]
```

The least-squares slope for `Y = c1 * X` using the points (1,3), (2,5), and (3,7), and weights 0.4, 0.9, and 0.4 is calculated below. The approximation of y-values and standard error about **x** = 1, 2, and 3 follows.

```
> let c1:cov11:sumsq:_ = wmul {1,2,3} {0.4,0.9,0.4} {3,5,7};
> mul_est 1 c1 cov11;
[2.44736842105263,0.362738125055006]
> mul_est 2 c1 cov11;
[4.89473684210526,0.725476250110012]
> mul_est 3 c1 cov11;
[7.34210526315789,1.08821437516502]
```

15.5 Statistics

This module is loaded via the command `using gsl::stats` and provides Pure wrappers for the GSL Statistics routines found in Chapter 20 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Statistics.html.

15.5.1 Routines

`gsl::stats::`**`mean`** data::matrix
implements `gsl_stats_mean` without `stride` and `n` arguments.

`gsl::stats::`**`variance`** data::matrix
implements `gsl_stats_variance` without `stride` and `n` arguments.

`gsl::stats::`**`variance`** data::matrix mean
implements `gsl_stats_variance_m` without `stride` and `n` arguments.

`gsl::stats::`**`sd`** data::matrix
implements `gsl_stats_sd` without `stride` and `n` arguments.

`gsl::stats::`**`sd_m`** data::matrix mean
implements `gsl_stats_sd_m` without `stride` and `n` arguments.

`gsl::stats::`**`tss`** data::matrix
implements `gsl_stats_tss` without `stride` and `n` arguments.

`gsl::stats::`**`tss_m`** data::matrix mean
implements `gsl_stats_tss_m` without `stride` and `n` arguments.

`gsl::stats::`**`variance_with_fixed_mean`** data::matrix mean
implements `gsl_stats_variance_with_fixed_mean` without `stride` and `n` arguments.

`gsl::stats::`**`sd_with_fixed_mean`** data::matrix mean
implements `gsl_stats_sd_with_fixed_mean` without `stride` and `n` arguments.

`gsl::stats::`**`absdev`** data::matrix
implements `gsl_stats_absdev` without `stride` and `n` arguments.

`gsl::stats::`**`absdev_m`** data::matrix mean
implements `gsl_stats_absdev_m` without `stride` and `n` arguments.

`gsl::stats::`**`skew`** data::matrix mean
implements `gsl_stats_skew` without `stride` and `n` arguments.

`gsl::stats::`**`skew_m_sd`** data::matrix mean sd
implements `gsl_stats_skew_m_sd` without `stride` and `n` arguments.

`gsl::stats::`**`kurtosis`** data::matrix
implements `gsl_stats_kurtosis` without `stride` and `n` arguments.

`gsl::stats::`**`kurtosis_m_sd`** data::matrix mean sd
implements `gsl_stats_kurtosis_m_sd` without `stride` and `n` arguments.

`gsl::stats::`**`lag1_autocorrelation`** data::matrix
implements `gsl_stats_lag1_autocorrelation` without `stride` and `n` arguments.

`gsl::stats::`**`lag1_autocorrelation_m`** data::matrix mean
implements `gsl_stats_lag1_autocorrelation_m` without `stride` and `n` arguments.

`gsl::stats::`**`covariance`** d1::matrix d2::matrix
implements `gsl_stats_covariance` without `stride1`, `stride2`, and `n` arguments.

`gsl::stats::`**`covariance_m`** d1::matrix d2::matrix mean1 mean2
implements `gsl_stats_covariance_m` without `stride1`, `stride2`, and `n` arguments.

`gsl::stats::`**`correlation`** d1::matrix d2::matrix
implements `gsl_stats_correlation` without `stride1`, `stride2`, and `n` arguments.

`gsl::stats::`**`wmean`** weight::matrix data::matrix
implements `gsl_stats_wmean` without `stride` and `n` arguments.

`gsl::stats::`**`wvariance`** weight::matrix data::matrix
implements `gsl_stats_wvariance` without `stride` and `n` arguments.

`gsl::stats::`**`wvariance_m`** weight::matrix data::matrix mean
implements `gsl_stats_wvariance_m` without `stride` and `n` arguments.

`gsl::stats::`**`wsd`** weight::matrix data::matrix
implements `gsl_stats_wsd` without `stride` and `n` arguments.

`gsl::stats::`**`wsd_m`** weight::matrix data::matrix mean
implements `gsl_stats_wsd_m` without `stride` and `n` arguments.

`gsl::stats::`**`wvariance_with_fixed_mean`** weight::matrix data::matrix mean
implements `gsl_stats_wvariance_with_fixed_mean` without `stride` and `n` arguments.

`gsl::stats::`**`wsd_with_fixed_mean`** weight::matrix data::matrix mean
implements `gsl_stats_wsd_with_fixed_mean` without `stride` and `n` arguments.

`gsl::stats::`**`wtss`** weight::matrix data::matrix
implements `gsl_stats_wtss` without `stride` and `n` arguments.

`gsl::stats::`**`wtss_m`** weight::matrix data::matrix mean
implements `gsl_stats_wtss_m` without `stride` and `n` arguments.

`gsl::stats::`**`wabsdev`** weight::matrix data::matrix
implements `gsl_stats_wabsdev` without `stride` and `n` arguments.

`gsl::stats::`**`wabsdev_m`** weight::matrix data::matrix mean
implements `gsl_stats_wabsdev_m` without `stride` and `n` arguments.

`gsl::stats::`**`wskew`** weight::matrix data::matrix
implements `gsl_stats_wskew` without `stride` and `n` arguments.

`gsl::stats::`**`wskew_m_sd`** weight::matrix data::matrix mean sd
implements `gsl_stats_wskew_m_sd` without `stride` and `n` arguments.

`gsl::stats::`**`wkurtosis`** weight::matrix data::matrix
implements `gsl_stats_wkurtosis` without `stride` and `n` arguments.

`gsl::stats::`**`wkurtosis_m_sd`** weight::matrix data::matrix
implements `gsl_stats_wkurtosis_m_sd` without `stride` and `n` arguments.

`gsl::stats::`**`max`** data::matrix
: implements `gsl_stats_max` without `stride` and `n` arguments.

`gsl::stats::`**`min`** data::matrix
: implements `gsl_stats_min` without `stride` and `n` arguments.

`gsl::stats::`**`minmax`** data::matrix
: implements `gsl_stats_minmax` without `stride` and `n` arguments. Results are returned as a list `[min, max]`.

`gsl::stats::`**`min_index`** data::matrix
: implements `gsl_stats_min_index` without `stride` and `n` arguments.

`gsl::stats::`**`max_index`** data::matrix
: implements `gsl_stats_max_index` without `stride` and `n` arguments.

`gsl::stats::`**`minmax_index`** data::matrix
: implements `gsl_stats_minmax_index` without `stride` and `n` arguments. Results are returned as a list `[min_index, max_index]`.

`gsl::stats::`**`median_from_sorted_data`** data::matrix
: implements `gsl_stats_median_from_sorted_data` without `stride` and `n` arguments.

`gsl::stats::`**`quantile_from_sorted_data`** data::matrix f::double
: implements `gsl_stats_quantile_from_sorted_data` without `stride` and `n` arguments.

15.5.2 Examples

The following illustrates the use of each function in the `stats` module.

```
> using gsl::stats;
> using namespace gsl::stats;
> mean {1,2,3,4,5};
3.0
> variance {1,2,3,4,5};
2.5
> variance_m {1,2,3,4,5}  4;
3.75
> sd {1,2,3,4,5};
1.58113883008419
> sd_m {1,2,3,4,5} 4;
1.93649167310371
> tss {1,2,3,4,5};
10.0
> tss_m {1,2,3,4,5} 4;
15.0
> variance_with_fixed_mean {0.0,1.2,3.4,5.6,6.0} 4.1;
6.314
> sd_with_fixed_mean {0.0,1.2,3.4,5.6,6.0} 4.1;
2.51276739870606
> absdev {2,2,3,4,4};
```

```
0.8
> absdev_m {2,2,3,4,4} 4;
1.0
> skew {1,1,1,1,2,2,2,2,2,2,2,2,3,30};
2.94796699504537
> skew_m_sd {1,2,2,3,3,3,3,3,3,3,4,4,5} 3 1;
0.0
> kurtosis  {1,2,2,3,3,3,3,3,3,3,4,4,5};
-0.230769230769231
> kurtosis_m_sd {1,2,2,3,3,3,3,3,3,3,4,4,5} 3 1;
-0.230769230769231
> lag1_autocorrelation {1,2,3,4,5};
0.4
> lag1_autocorrelation_m {1,2,3,4,5} 2.5;
0.444444444444444
> covariance {1,2,3,4,5} {3.0,4.5,6.0,7.5,9.0};
3.75
> covariance_m {1,2,3,4,5} {3.0,4.5,6.0,7.5,9.0} 3 6;
3.75
> correlation {1,2,3,4} {2,3,4,5};
1.0
> wmean {0.4,0.2,0.3,0.3,0.3} {2,3,4,5,6};
3.93333333333333
> wvariance {0.4,0.2,0.3,0.3,0.3} {2,3,4,5,6};
2.7752808988764
> wvariance_m {0.4,0.2,0.3,0.3,0.3} {2,3,4,5,6} 3.0;
3.87640449438202
> wsd {0.4,0.2,0.3,0.3,0.3} {2,3,4,5,6};
1.66591743459164
> wsd_m {0.4,0.2,0.3,0.3,0.3} {2,3,4,5,6} 3.0;
1.96885867811329
> wvariance_with_fixed_mean {1,2,3,4} {1,2,3,4} 2.5;
1.25
> wsd_with_fixed_mean {1,2,3,4} {1,2,3,4} 2.5;
1.11803398874989
> wtss {1,1,2,2} {2,3,4,5};
6.83333333333333
> wtss_m {1,1,2,2} {2,3,4,5} 3.1;
10.06
> wabsdev {1,1,2,2} {2,3,4,5};
0.888888888888889
> wabsdev_m {1,1,2,2} {2,3,4,5} 3.1;
1.13333333333333
> wskew {1,1,2,2} {2,3,4,5};
-0.299254338484713
> wskew_m_sd {1,1,2,2} {2,3,4,5} 3.1 1.2;
1.33526234567901
> wkurtosis {1,1,2,2} {2,3,4,5};
-1.96206512878137
> wkurtosis_m_sd {1,1,2,2} {2,3,4,5} 3.1 1.2;
-0.681921939300412
> min {9,4,2,1,9};
```

```
1
> max {9.1,4.2,2.6,1.1,9.2};
9.2
> minmax {9.0,4.0,2.0,1.0,9.0};
[1.0,9.0]
> min_index {9.1,4.2,2.6,1.1,9.2};
3
> max_index {9,4,2,1,9};
0
> minmax_index {9,4,2,1,0,9};
[4,0]
> median_from_sorted_data {1.0,2.0,3.0};
2.0
> quantile_from_sorted_data {1.0,2.0,3.0} 0.25;
1.5
```

15.6 Random Number Distributions

This module is loaded via the command `using gsl::randist` and provides Pure wrappers for the GSL random distribution routines found in Chapter 19 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Random-Number-Distributions.html.

There are two namespaces provided by randist.pure, `gsl::ran` for probability densitity functions and `gsl::cdf` for cumulative distribution functions. The two namespaces minimize typing of the prefixes `gsl_ran_` and `gsl_cdf_` respectively.

15.6.1 Routines

`gsl::ran::`**`ugaussian_pdf`** x
: implements `gsl_ran_ugaussian`.

`gsl::ran::`**`gaussian_pdf`** x sigma
: implements `gsl_ran_gaussian_pdf`.

`gsl::ran::`**`gaussian_tail_pdf`** x a sigma
: implements `gsl_ran_gaussian_tail_pdf`.

`gsl::ran::`**`ugaussian_tail_pdf`** x a
: implements `gsl_ran_ugaussian_tail_pdf`.

`gsl::ran::`**`bivariate_gaussian_pdf`** x a
: implements `gsl_ran_bivariate_gaussian_pdf`.

`gsl::ran::`**`exponential_pdf`** x mu
: implements `gsl_ran_exponential_pdf`.

`gsl::ran::`**`laplace_pdf`** x a
implements `gsl_ran_laplace_pdf`.

`gsl::ran::`**`exppow_pdf`** x a b
implements `gsl_ran_exppow_pdf`.

`gsl::ran::`**`cauchy_pdf`** x a
implements `gsl_ran_cauchy_pdf`.

`gsl::ran::`**`rayleigh_pdf`** x sigma
implements `gsl_ran_rayleigh_pdf`.

`gsl::ran::`**`rayleigh_tail_pdf`** x a sigma
implements `gsl_ran_rayleigh_tail_pdf`.

`gsl::ran::`**`landau_pdf`** x
implements `gsl_ran_landau_pdf`.

`gsl::ran::`**`gamma_pdf`** x a b
implements `gsl_ran_gamma_pdf`.

`gsl::ran::`**`flat_pdf`** x a b
implements `gsl_ran_flat_pdf`.

`gsl::ran::`**`lognormal_pdf`** x zeta sigma
implements `gsl_ran_lognormal_pdf`.

`gsl::ran::`**`chisq_pdf`** x nu
implements `gsl_ran_chisq_pdf`.

`gsl::ran::`**`fdist_pdf`** x nu1 nu2
implements `gsl_ran_fdist_pdf`.

`gsl::ran::`**`tdist_pdf`** x nu
implements `gsl_ran_tdist_pdf`.

`gsl::ran::`**`beta_pdf`** x a b
implements `gsl_ran_beta_pdf`.

`gsl::ran::`**`logistic_pdf`** x a
implements `gsl_ran_logistic_pdf`.

`gsl::ran::`**`pareto_pdf`** x a b
implements `gsl_ran_pareto_pdf`.

`gsl::ran::`**`weibull_pdf`** x a b
implements `gsl_ran_weibull_pdf`.

`gsl::ran::`**`gumbel1_pdf`** x a b
implements `gsl_ran_gumbel1_pdf`.

`gsl::ran::`**`gumbel2_pdf`** x a b
implements `gsl_ran_gumbel2_pdf`.

`gsl::ran::`**`dirichlet_pdf`** alpha::matrix theta::matrix
implements `gsl_ran_dirichlet_pdf`.

`gsl::ran::`**`dirichlet_lnpdf`** alpha::matrix theta::matrix
implements `gsl_ran_dirichlet_lnpdf`.

`gsl::ran::`**`discrete_preproc`** p::matrix
implements `gsl_ran_discrete_preproc` without the `K` parameter.

`gsl::ran::`**`discrete_pdf`** k::int p::pointer
implements `gsl_ran_discrete_pdf` without the `K` parameter.

`gsl::ran::`**`discrete_free`** p::pointer
implements `gsl_ran_discrete_free`

`gsl::ran::`**`poisson_pdf`** k::int mu
implements `gsl_ran_poisson_pdf`.

`gsl::ran::`**`bernoulli_pdf`** k::int p
implements `gsl_ran_bernoulli_pdf`.

`gsl::ran::`**`binomial_pdf`** k::int p n::int
implements `gsl_ran_binomial_pdf`.

`gsl::ran::`**`multinomial_pdf`** p::matrix n::matrix
implements `gsl_ran_multinomial_pdf`.

`gsl::ran::`**`multinomial_lnpdf`** p::matrix n::matrix
implements `gsl_ran_multinomial_lnpdf`.

`gsl::ran::`**`negative_binomial_pdf`** k::int p n
implements `gsl_ran_negative_binomial_pdf`.

`gsl::ran::`**`pascal_pdf`** k::int p n::int
implements `gsl_ran_pascal_pdf`.

`gsl::ran::`**`geometric_pdf`** k::int p
implements `gsl_ran_geometric_pdf`.

`gsl::ran::`**`hypergeometric_pdf`** k::int n1::int n2::int t::int
implements `gsl_ran_hypergeometric_pdf`.

`gsl::ran::`**`logarithmic_pdf`** k::int p
implements `gsl_ran_logarithmic_pdf`.

`gsl::cdf::`**`ugaussian_P`** x
implements `gsl_cdf_ugaussian_P`.

`gsl::cdf::`**`ugaussian_Q`** x
implements `gsl_cdf_ugaussian_Q`.

`gsl::cdf::`**`ugaussian_Pinv`** p
implements `gsl_cdf_ugaussian_Pinv`.

`gsl::cdf::`**`ugaussian_Qinv`** q
implements `gsl_cdf_ugaussian_Qinv`.

`gsl::cdf::`**`gaussian_P`** x sigma
implements `gsl_cdf_gaussian_P`.

gsl::cdf::**gaussian_Q** x sigma
implements gsl_cdf_gaussian_Q.

gsl::cdf::**gaussian_Pinv** p sigma
implements gsl_cdf_gaussian_Pinv.

gsl::cdf::**guassian_Qinv** q sigma
implements gsl_cdf_gaussian_Qinv.

gsl::cdf::**exponential_P** x mu
implements gsl_cdf_exponential_P.

gsl::cdf::**exponential_Q** x mu
implements gsl_cdf_exponential_Q.

gsl::cdf::**exponential_Pinv** p mu
implements gsl_cdf_exponential_Pinv.

gsl::cdf::**exponential_Qinv** q mu
implements gsl_cdf_exponential_Qinv.

gsl::cdf::**laplace_P** x a
implements gsl_cdf_laplace_P.

gsl::cdf::**laplace_Q** x a
implements gsl_cdf_laplace_Q.

gsl::cdf::**laplace_Pinv** p a
implements gsl_cdf_laplace_Pinv.

gsl::cdf::**laplace_Qinv** q a
implements gsl_cdf_laplace_Qinv.

gsl::cdf::**exppow_P** x a b
implements gsl_cdf_exppow_P.

gsl::cdf::**exppow_Q** x a b
implements gsl_cdf_exppow_Q.

gsl::cdf::**cauchy_P** x a
implements gsl_cdf_cauchy_P.

gsl::cdf::**cauchy_Q** x a
implements gsl_cdf_cauchy_Q.

gsl::cdf::**cauchy_Pinv** p a
implements gsl_cdf_cauchy_Pinv.

gsl::cdf::**cauchy_Qinv** q a
implements gsl_cdf_cauchy_Qinv.

gsl::cdf::**rayleigh_P** x sigma
implements gsl_cdf_rayleigh_P.

gsl::cdf::**rayleigh_Q** x sigma
implements gsl_cdf_rayleigh_Q.

`gsl::cdf::`**`rayleigh_Pinv`** p sigma
implements `gsl_cdf_rayleigh_Pinv`.

`gsl::cdf::`**`rayleigh_Qinv`** q sigma
implements `gsl_cdf_rayleigh_Qinv`.

`gsl::cdf::`**`gamma_P`** x a b
implements `gsl_cdf_gamma_P`.

`gsl::cdf::`**`gamma_Q`** x a b
implements `gsl_cdf_gamMa_Q`.

`gsl::cdf::`**`gamma_Pinv`** p a b
implements `gsl_cdf_gamma_Pinv`.

`gsl::cdf::`**`gamma_Qinv`** q a b
implements `gsl_cdf_gamma_Qinv`.

`gsl::cdf::`**`flat_P`** x a b
implements `gsl_cdf_flat_P`.

`gsl::cdf::`**`flat_Q`** x a b
implements `gsl_cdf_flat_Q`.

`gsl::cdf::`**`flat_Pinv`** p a b
implements `gsl_cdf_flat_Pinv`.

`gsl::cdf::`**`flat_Qinv`** q a b
implements `gsl_cdf_flat_Qinv`.

`gsl::cdf::`**`lognormal_P`** x zeta sigma
implements `gsl_cdf_lognormal_P`.

`gsl::cdf::`**`lognormal_Q`** x zeta sigma
implements `gsl_cdf_lognormal_Q`.

`gsl::cdf::`**`lognormal_Pinv`** p zeta sigma
implements `gsl_cdf_lognormal_Pinv`.

`gsl::cdf::`**`lognormal_Qinv`** q zeta sigma
implements `gsl_cdf_lognormal_Qinv`.

`gsl::cdf::`**`chisq_P`** x nu
implements `gsl_cdf_chisq_P`.

`gsl::cdf::`**`chisq_Q`** x nu
implements `gsl_cdf_chisq_Q`.

`gsl::cdf::`**`chisq_Pinv`** p nu
implements `gsl_cdf_chisq_Pinv`.

`gsl::cdf::`**`chisq_Qinv`** q nu
implements `gsl_cdf_chisq_Qinv`.

`gsl::cdf::`**`fdist_P`** x nu1 nu2
implements `gsl_cdf_fdist_P`.

`gsl::cdf::`**`fdist_Q`** x nu1 nu2
implements `gsl_cdf_fdist_Q`.

`gsl::cdf::`**`fdist_Pinv`** p nu1 nu2
implements `gsl_cdf_fdist_Pinv`.

`gsl::cdf::`**`fdist_Qinv`** q nu1 nu2
implements `gsl_cdf_fdist_Qinv`.

`gsl::cdf::`**`tdist_P`** x nu
implements `gsl_cdf_tdist_P`.

`gsl::cdf::`**`tdist_Q`** x nu
implements `gsl_cdf_tdist_Q`.

`gsl::cdf::`**`tdist_Pinv`** p nu
implements `gsl_cdf_tdist_Pinv`.

`gsl::cdf::`**`tdist_Qinv`** q nu
implements `gsl_cdf_tdist_Qinv`.

`gsl::cdf::`**`beta_P`** x a b
implements `gsl_cdf_beta_P`.

`gsl::cdf::`**`beta_Q`** x a b
implements `gsl_cdf_beta_Q`.

`gsl::cdf::`**`beta_Pinv`** p a b
implements `gsl_cdf_beta_Pinv`.

`gsl::cdf::`**`beta_Qinv`** q a b
implements `gsl_cdf_beta_Qinv`.

`gsl::cdf::`**`logistic_P`** x a
implements `gsl_cdf_logistic_P`.

`gsl::cdf::`**`logistic_Q`** x a
implements `gsl_cdf_logistic_Q`.

`gsl::cdf::`**`logistic_Pinv`** p a
implements `gsl_cdf_logistic_Pinv`.

`gsl::cdf::`**`logistic_Qinv`** q a
implements `gsl_cdf_logistic_Qinv`.

`gsl::cdf::`**`pareto_P`** x a b
implements `gsl_cdf_pareto_P`.

`gsl::cdf::`**`pareto_Q`** x a b
implements `gsl_cdf_pareto_Q`.

`gsl::cdf::`**`pareto_Pinv`** p a b
implements `gsl_cdf_pareto_Pinv`.

`gsl::cdf::`**`pareto_Qinv`** q a b
implements `gsl_cdf_pareto_Qinv`.

gsl::cdf::**weibull_P** x a b
implements gsl_cdf_weibull_P.

gsl::cdf::**weibull_Q** x a b
implements gsl_cdf_weibull_Q.

gsl::cdf::**weibull_Pinv** p a b
implements gsl_cdf_weibull_Pinv.

gsl::cdf::**weibull_Qinv** q a b
implements gsl_cdf_weibull_Qinv.

gsl::cdf::**gumbel1_P** x a b
implements gsl_cdf_gumbel1_P.

gsl::cdf::**gumbel1_Q** x a b
implements gsl_cdf_gumbel1_Q.

gsl::cdf::**gumbel1_Pinv** p a b
implements gsl_cdf_gumbel1_Pinv.

gsl::cdf::**gumbel1_Qinv** q a b
implements gsl_cdf_gumbel1_Qinv.

gsl::cdf::**gumbel2_P** x a b
implements gsl_cdf_gumbel2_P.

gsl::cdf::**gumbel2_Q** x a b
implements gsl_cdf_gumbel2_Q.

gsl::cdf::**gumbel2_Pinv** p a b
implements gsl_cdf_gumbel2_Pinv.

gsl::cdf::**gumbel2_Qinv** q a b
implements gsl_cdf_gumbel2_Qinv.

gsl::cdf::**poisson_P** k::int mu
implements gsl_cdf_poisson_P.

gsl::cdf::**poisson_Q** k::int mu
implements gsl_cdf_poisson_Q.

gsl::cdf::**binomial_P** k::int p n::int
implements gsl_cdf_binomial_P.

gsl::cdf::**binomial_Q** k::int q n::int
implements gsl_cdf_binomial_Q.

gsl::cdf::**negative_binomial_P** k::int p n
implements gsl_cdf_negative_binomial_P.

gsl::cdf::**negative_binomial_Q** k::int p n
implements gsl_cdf_negative_binomial_Q.

gsl::cdf::**pascal_P** k::int p n::int
implements gsl_cdf_pascal_P.

`gsl::cdf::`**`pascal_Q`** k::int p n::int
: implements `gsl_cdf_pascal_Q`.

`gsl::cdf::`**`geometric_P`** k::int p
: implements `gsl_cdf_geometric_P`.

`gsl::cdf::`**`geometric_Q`** k::int p
: implements `gsl_cdf_geometric_Q`.

`gsl::cdf::`**`hypergeometric_P`** k::int n1::int n2::int t::int
: implements `gsl_cdf_hypergeometric_P`.

`gsl::cdf::`**`hypergeometric_Q`** k::int n1::int n2::int t::int
: implements `gsl_cdf_hypergeometric_Q`.

15.6.2 Examples

The following illustrates the use of each function in the `randist` module. The pdf functions are illustrated first.

```
> using gsl::stats;
> using namespace gsl::ran;
> ugaussian_pdf 1.2;
0.194186054983213
> gaussian_pdf (-1.3) 1.5;
0.182690978264686
> gaussian_tail_pdf 2.0 1.0 1.5;
0.433042698395299
> ugaussian_tail_pdf 2.0 1.0;
0.34030367841782
> bivariate_gaussian_pdf 1.2 0.9 1.0 1.0 0.95;
0.184646843689817
> exponential_pdf 1.0 0.5;
0.270670566473225
> laplace_pdf 1.5 2.0;
0.118091638185254
> exppow_pdf 0.0 1.0 1.5;
0.553866083716236
> cauchy_pdf (-1.0) 1.0;
0.159154943091895
> rayleigh_pdf 2.5 1.0;
0.109842334058519
> rayleigh_tail_pdf 1.5 1.0 1.0;
0.802892142778485
> landau_pdf 1.1;
0.140968737919623
> gamma_pdf 1.0 1.0 1.5;
0.342278079355061
> flat_pdf 1.0 0.5 2.5;
0.5
> lognormal_pdf 0.01 0.0 1.0;
0.000990238664959182
```

```
> chisq_pdf 1.0 2.0;
0.303265329856317
> fdist_pdf 0.5 3.0 2.0;
0.480970043785452
> tdist_pdf 0.1 10.0;
0.386975225815181
> beta_pdf 0.5 4.0 1.0;
0.499999999999999
> logistic_pdf (-1.0) 2.0;
0.117501856100797
> pareto_pdf 0.01 3.0 2.0;
0.0
> weibull_pdf  0.01 1.0 1.0;
0.990049833749168
> gumbel1_pdf 0.01 1.0 1.0;
0.367861108816436
> gumbel2_pdf 0.01 1.0 1.0;
3.72007597602084e-40
> dirichlet_pdf {0.1,0.2,0.8} {2.0,2.0,2.0};
0.00501316294425874
> dirichlet_lnpdf {0.1,0.2,0.8} {2.0,2.0,2.0};
-5.29568823688856
> poisson_pdf 4 0.4;
0.000715008049104682
> bernoulli_pdf 1 0.7;
0.7
> binomial_pdf 3 0.5 9;
0.1640625
> multinomial_pdf {0.1,0.2,0.7} {2,2,2};
0.0
> multinomial_lnpdf {0.1,0.2,0.7} {2,2,2};
-1728120799.71174
> negative_binomial_pdf 10 0.5 3.5;
0.0122430486923836
> pascal_pdf 10 0.5 3;
0.00805664062499999
> geometric_pdf 5 0.4;
0.05184
> hypergeometric_pdf 1 5 20 3;
0.413043478260872
> logarithmic_pdf 10 0.7;
0.00234619293712492
> test_discrete
>   = v
>     when
>       px = discrete_preproc {0.1,0.3,0.4};
>       v = discrete_pdf 0 px +
>           discrete_pdf 1 px +
>           discrete_pdf 2 px;
>       _ = discrete_free px
>     end;
> test_discrete;
```

```
1.0
```

The cumulative distribution functions are shown.

```
> using namespace gsl::cdf;
> ugaussian_P  (-1.3);
0.0968004845856103
> ugaussian_Q  (-1.3);
0.90319951541439
> ugaussian_Pinv  0.84;
0.994457883209753
> ugaussian_Qinv  0.84;
-0.994457883209753
> gaussian_P  (1.3)  1.5;
0.806937662858093
> gaussian_Q  (1.3)  1.5;
0.193062337141907
> gaussian_Pinv  0.4  5.0;
-1.266735515679
> gaussian_Qinv  0.4 5.0;
1.266735515679
> exponential_P  1.0  0.5;
0.864664716763387
> exponential_Q  1.0  0.5;
0.135335283236613
> exponential_Pinv  0.6  0.5;
0.458145365937077
> exponential_Qinv  0.6  0.5;
0.255412811882995
> laplace_P  1.5  2.0;
0.763816723629493
> laplace_Q  1.5  2.0;
0.236183276370507
> laplace_Pinv  0.6  2.0;
0.446287102628419
> laplace_Qinv  0.4  2.0;
0.446287102628419
> exppow_P  0.0  1.0  2.5;
0.5
> exppow_Q  0.0  1.0  0.5;
0.5
> cauchy_P  (-1.0)  1.0;
0.25
> cauchy_Q  (-1.0)  1.0;
0.75
> cauchy_Pinv  0.75  1.0;
1.0
> cauchy_Qinv  0.25  1.0;
1.0
> rayleigh_P  1.5  2.0;
0.245160398010993
> rayleigh_Q  0.5  1.0;
```

```
0.882496902584595
> rayleigh_Pinv  0.5  1.0;
1.17741002251547
> rayleigh_Qinv  0.5  1.0;
1.17741002251547
> gamma_P  1.0  1.0  3.0;
0.283468689426211
> gamma_Q  1.0  1.0  3.0;
0.716531310573789
> gamma_Pinv  0.5  1.0  1.0;
0.693147180559945
> gamma_Qinv  0.5  1.0  1.0;
0.693147180559945
> flat_P  2.0  1.2  4.8;
0.222222222222222
> flat_Q  2.0  1.2  4.8;
0.777777777777778
> flat_Pinv  0.2  0.5  2.5;
0.9
> flat_Qinv  0.2  0.5  2.5;
2.1
> lognormal_P  0.01  0.0  1.0;
2.06064339597172e-06
> lognormal_Q  0.01  0.0  1.0;
0.999997939356604
> lognormal_Pinv  0.1  0.0  1.0;
0.27760624185201
> lognormal_Qinv  0.1  0.0  1.0;
3.60222447927916
> chisq_P  1.0  2.0;
0.393469340287367
> chisq_Q  1.0  2.0;
0.606530659712633
> chisq_Pinv  0.5  2.0;
0.221199216928595
> chisq_Qinv  0.5  2.0;
1.38629436111989
> fdist_P  1.0  3.0  2.0;
0.46475800154489
> fdist_Q  1.0  3.0  2.0;
0.53524199845511
> fdist_Pinv  0.5  3.0  2.0;
1.13494292261288
> fdist_Qinv  0.5  3.0  2.0;
1.13494292261288
> tdist_P  2.1  10.0;
0.968961377898891
> tdist_Q  (-2.1)  10.0;
0.968961377898891
> tdist_Pinv  0.68  10.0;
0.482264205919689
> tdist_Qinv  0.68  10.0;
```

```
-0.482264205919689
> beta_P  0.75  2.0  2.0;
0.84375
> beta_Q  0.75  2.0  2.0;
0.15625
> beta_Pinv  0.75  2.0  2.0;
0.673648177666931
> beta_Qinv  0.25  2.0  2.0;
0.673648177666931
> logistic_P  (-1.0)  2.0;
1
> logistic_Q  (-1.0)  2.0;
0.622459331201855
> logistic_Pinv  0.75  1.0;
1.09861228866811
> logistic_Qinv  0.25  1.0;
1.09861228866811
> pareto_P  2.01  3.0  2.0;
0.0148512406901899
> pareto_Q  2.01  3.0  2.0;
0.98514875930981
> pareto_Pinv  0.1  3.0  2.0;
2.07148833730257
> pareto_Qinv  0.1  3.0  2.0;
4.30886938006377
> weibull_P  1.01  1.0  2.0;
0.639441117518024
> weibull_Q  1.01  2.0  3.0;
0.879160657465162
> weibull_Pinv  0.1  1.0  2.0;
0.324592845974501
> weibull_Qinv  0.1  1.0  2.0;
1.51742712938515
> gumbel1_P  1.01  1.0  1.0;
0.694739044426344
> gumbel1_Q  1.01  1.0  1.0;
0.305260955573656
> gumbel1_Pinv  0.1  1.0  1.0;
-0.834032445247956
> gumbel1_Qinv  0.1  1.0  1.0;
2.25036732731245
> gumbel2_P  1.01  1.0  1.0;
0.371539903071873
> gumbel2_Q  1.01  1.0  1.0;
0.628460096928127
> gumbel2_Pinv  0.1  1.0  1.0;
0.434294481903252
> gumbel2_Qinv  0.1  1.0  1.0;
9.4912215810299
> poisson_P  4  0.4;
0.999938756672898
> poisson_Q  4  0.6;
```

```
0.000394486018340255
> binomial_P  3  0.5  10;
0.171874999999999
> binomial_Q  3  0.5  10;
0.828125000000001
> negative_binomial_P  10  0.5  3.0;
0.98876953125
> negative_binomial_Q  10  0.5  3.0;
0.01123046875
> pascal_P  10  0.5  3;
0.98876953125
> pascal_Q  10  0.5  3;
0.01123046875
> geometric_P  5  0.4;
0.92224
> geometric_Q  5  0.6;
0.01024
> hypergeometric_P  1  5  20  3;
0.908695652173913
> hypergeometric_Q  1  5  20  3;
0.0913043478260873
```

15.7 Sorting

This module is loaded via the command `using gsl::sort` and provides Pure wrappers for the GSL sorting routines found in Chapter 11 of the GSL manual,

http://www.gnu.org/software/gsl/manual/html_node/Sorting.html.

15.7.1 Routines

`gsl::`**`sort_vector`** m::matrix
: implements `gsl_sort` and `gsl_sort_int` without `stride` and `n` parameters.

`gsl::`**`sort_vector_index`** m::matrix
: implements `gsl_sort_index` and `gsl_sort_int_index` without `stride` and `n` parameters.

15.7.2 Examples

Usage of each library routine is illustrated below.

```
> using gsl::sort;
> using namespace gsl;
> sort_vector {0,3,2,4,5};
{0,2,3,4,5}
```

```
> sort_vector_index {0.0,1.0,5.0,2.0,8.0,0.0};
{5,0,1,3,2,4}
```

Chapter 16

pure-mpfr

Version 0.4, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

The GNU MPFR library is a C library for multiple-precision floating-point computations with correct rounding. It is based on GMP which Pure also uses for its bigint support.

This module makes the MPFR multiprecision floats (henceforth referred to as `mpfr` numbers or values) available in Pure, so that they work with the other types of Pure numbers in an almost seamless fashion. Pure `mpfr` values are represented as pointers which can readily be passed as arguments to the MPFR functions, so the representation only involves minimal overhead on the Pure side.

The module defines the type of `mpfr` values as an instance of Pure's `real` type, so that it becomes a well-behaved citizen of Pure's numeric tower. Memory management of these values is automatic. You can create an `mpfr` value from any other kind of Pure real value (`int`, `bigint` or `double`), or from a string in decimal notation, using the `mpfr` function. Back conversions are provided from `mpfr` to `int`, `bigint`, `double` and `string` (the latter by means of a custom pretty-printer installed by this module, so that mpfr values are printed in a format similar to the `printf %g` format). Integration with Pure's `complex` type is provided as well.

Please note that this module needs more testing and the API hasn't been finalized yet, but it should be perfectly usable already. As usual, please report any bugs on the Pure issue tracker, on the Pure mailing list, or directly to the author, see http://pure-lang.googlecode.com/.

16.1 Copying

Copyright (c) 2011 by Albert Graef.

pure-mpfr is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

16.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-mpfr-0.4.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure and libmpfr installed.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually, please check the Makefile for details.

Note: This module requires Pure 0.50 or later and libmpfr 3.x (3.0.0 has been tested). Older libmpfr versions (2.x) probably require some work.

16.3 Usage

After installation, you can use the operations of this module by placing the following import declaration in your Pure programs:

```
using mpfr;
```

Note: This also pulls in the `math` standard library module, whose operations are overloaded by the `mpfr` module in order to provide support for `mpfr` values. Thus you don't need to explicitly import the `math` module when using the `mpfr` module.

If you use both the `mpfr` module and the `pointers` standard library module in your script, make sure that you import the `pointers` module *after* `mpfr`, so that the definitions of pointer arithmetic in the `pointers` module do not interfere with the overloading of arithmetic operations in the `mpfr` module.

16.3.1 Precision and Rounding

The following operations of the MPFR library are provided to inspect and change the default precision and rounding modes used by MPFR.

mpfr_get_default_prec
mpfr_set_default_prec prec
: Get and set the default precision in terms of number of bits in the mantissa, including the sign. MPFR initially sets this to 53 (matching the mantissa size of `double` values). It can be changed to any desired value not less than 2.

mpfr_get_prec x
: Get the precision of an `mpfr` number `x`. Note that `mpfr` numbers always keep the precision they were created with, but it is possible to create a new `mpfr` number with any given precision from an existing `mpfr` number using the `mpfr` function, see below.

mpfr_get_default_rounding_mode
mpfr_set_default_rounding_mode rnd
: Get and set the default rounding mode, which is used for all arithmetic operations and mathematical functions provided by this module. The given rounding mode `rnd` must be one of the supported rounding modes listed below.

constant MPFR_RNDN // round to nearest, with ties to even
constant MPFR_RNDZ // round toward zero
constant MPFR_RNDU // round toward +Inf
constant MPFR_RNDD // round toward -Inf
constant MPFR_RNDA // round away from zero
: Supported rounding modes. Please check the MPFR documentation for details.

In addition, the following operations enable you to control the precision in textual representations of `mpfr` values. This information is used by the custom pretty-printer for `mpfr` values installed by the module.

mpfr_get_print_prec
mpfr_set_print_prec prec
: Get and set the precision (number of decimal digits in the mantissa) used by the pretty-printer.

16.3.2 MPFR Numbers

The module defines the following data type for representing `mpfr` values, which is a subtype of the Pure `real` type:

type mpfr
: This is a tagged pointer type (denoted `mpfr*` in Pure extern declarations) which is compatible with the `mpfr_t` and `mpfr_ptr` data types of the MPFR C library. Members of this type are "cooked" pointers, which are allocated dynamically and freed automatically when they are garbage-collected (by means of a corresponding Pure sentry).

mpfrp x
: Type predicate checking for `mpfr` values.

16.3.3 Conversions

The following operations are provided to convert between `mpfr` numbers and other kinds of Pure `real` values.

mpfr x
mpfr (x,prec)
mpfr (x,prec,rnd)
: This function converts any real number (`int`, `bigint`, `double`, `rational`, `mpfr`) to an `mpfr` value.

  Optionally, it is possible to specify a precision (number of bits in the mantissa) `prec` and a rounding mode `rnd` (one of the `MPFR_RND` constants), otherwise MPFR's default precision and rounding mode are used (see Precision and Rounding above). Note that this function may also be used to convert an `mpfr` to a new `mpfr` number, possibly with a different precision and rounding.

  The argument `x` can also be a string denoting a floating point number in decimal notation with optional sign, decimal point and/or scaling factor, which is parsed and converted to an `mpfr` number using the corresponding MPFR function.

int x
bigint x
double x
: Convert an `mpfr` number x to the corresponding type of real number. Please note that there is no `rational` conversion, as MPFR does not provide such an operation, but if you need this then you can first convert `x` to a `double` and then apply the standard library `rational` function to it (this may loose precision, of course).

str x
: By virtue of the custom pretty-printer provided by this module, the standard library `str` function can be used to obtain a printable representation of an `mpfr` number `x` in decimal notation. The result is a string.

floor x
ceil x
round x
trunc x
frac x
: Rounding and truncation functions. These all take and yield `mpfr` numbers. `frac` returns the fractional part of an `mpfr` number, i.e., `x-trunc x`.

16.3.4 Arithmetic

The following standard operators (see the *Pure Library Manual*) are overloaded to provide `mpfr` arithmetic and comparisons. These all handle mixed `mpfr`/`real` operands.

- x
x **+** y
x **-** y
x ***** y
x **/** y
x **^** y
: Arithmetic operations.

x **==** y
x **~=** y
x **<=** y
x **>=** y
x **<** y
x **>** y
: Comparisons.

16.3.5 Math Functions

The following functions from the `math` module are overloaded to provide support for `mpfr` values. Note that it is also possible to invoke the corresponding functions from the MPFR library in a direct fashion, using the same function names with an additional `_mpfr` suffix. These functions also accept other kinds of `real` arguments which are converted to `mpfr` before applying the MPFR function.

abs x
: Absolute value (this is implemented directly, so there's no corresponding `_mpfr` function for this).

sqrt x
exp x
ln x
log x
: Square root, exponential and logarithms.

sin x
cos x
tan x
asin x
acos x
atan x
atan2 y x
: Trigonometric functions.

sinh x

cosh x
tanh x
asinh x
acosh x
atanh x
Hyperbolic trigonometric functions.

16.3.6 Complex Number Support

The following functions from the `math` module are overloaded to provide support for complex values involving `mpfr` numbers:

complex x
polar x
rect x
cis x
arg x
re x
im x
conj x

16.4 Examples

Import the module and set the default precision:

```
> using mpfr;
> mpfr_set_default_prec 64; // extended precision (long double on x86)
()
```

Calculate pi with the current precision. Note that mixed arithmetic works with any combination of real and mpfr numbers.

```
> let Pi = 4*atan (mpfr 1);
> pi; Pi; abs (Pi-pi);
3.14159265358979
3.14159265358979323851
1.22514845490862001043e-16

> let Pi2 = Pi^2;
> Pi2; sqrt Pi2; sqrt Pi2 == Pi;
9.86960440108935861941
3.14159265358979323851
1
```

You can also query the precision of a number and change it on the fly:

```
> Pi; mpfr_get_prec Pi;
3.14159265358979323851
```

```
64
> let Pi1 = mpfr (Pi,53); Pi1; mpfr_get_prec Pi1;
3.1415926535897931
53
```

Complex `mpfr` numbers work, too:

```
> let z = mpfr 2^(1/i); z;
0.769238901363972126565+:-0.638961276313634801184
> let z = ln z/ln (mpfr 2); z;
0.0+:-1.0
> abs z, arg z;
1.0,-1.5707963267948966l926
> polar z;
1.0<:-1.5707963267948966l926
```

Chapter 17

pure-octave

Version 0.2, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

A Pure interface to GNU Octave.

17.1 Introduction

This is an Octave module for the Pure programming language, based on Paul Kienzle's octave_embed which allows Octave to be embedded in other languages. It allows you to execute arbitrary Octave commands and Octave functions from Pure.

17.2 Copying

pure-octave is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pure-octave is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Please see the accompanying COPYING file for the precise license terms. The GPL can also be read online at http://www.gnu.org/licenses/.

17.3 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-octave-0.2.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have both Pure and Octave installed (including Octave's `mkoctfile` utility and the corresponding header files and libraries).

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix. Please see the Makefile for details.

NOTE: This release of pure-gen has been tested with Octave 3.4.0. Older versions might require some fiddling with the sources to get the embedded Octave interface working.

17.4 Basic Usage

Import this module into your Pure scripts as follows:

```
using octave;
```

This will add an embedded instance of the Octave interpreter to your program. (You can import this module as often as you want, but there's always only one instance of Octave in each process.)

octave_eval s
: Executes arbitrary Octave code.

```
> octave_eval "eig([1 2;3 4])";
ans =

  -0.37228
   5.37228

0
```

This prints the result on stdout and returns a result code (zero if everything went fine). To suppress the printing of the result, simply terminate the Octave statement with a semicolon:

```
> octave_eval "eig([1 2;3 4]);";
0
```

octave_set var val
octave_get var
: Set and get global variables in the Octave interpreter.

This allows you to define values to be used when evaluating Octave code, and to transfer results back to Pure. However, before such globals can be accessed in Octave, you must explicitly declare them as globals:

```
> octave_eval "global x y ans";
0
```

Now you can use `octave_set` and `octave_get` to transfer values between Pure and Octave as follows:

```
> octave_set "x" {1.0,2.0;3.0,4.0};
{1.0,2.0;3.0,4.0}
> octave_eval "eig(x);";
0
> octave_get "ans";
{-0.372281323269014;5.37228132326901}
```

Note that the most recent result can always be accessed through Octave's `ans` variable. You can also use an explicit variable definition as follows:

```
> octave_eval "y = eig(x);";
0
> octave_get "y";
{-0.372281323269014;5.37228132326901}
```

17.5 Direct Function Calls

`octave_call` fun n args

Call an octave function in a direct fashion. `fun` denotes the name of the function, `n` the number of function results and `args` the function arguments.

```
> let x = {1.0,2.0;3.0,4.0};
> octave_call "eig" 1 x;
{-0.372281323269014;5.37228132326901}
```

Note the second argument, which denotes the desired number of *return* values. This will usually be 1 (or 0 if you don't care about the result), but some Octave functions may return a variable number of results, depending on how they're called. Multiple values are returned as tuples in Pure:

```
> octave_call "eig" 2 x;
{-0.824564840132394,-0.415973557919284;0.565767464968992,-0.909376709132124},
{-0.372281323269014,0.0;0.0,5.37228132326901}
```

If there are multiple *arguments*, you can specify them either as a tuple or a list:

```
> octave_call "norm" 1 (x, 2);
5.46498570421904
> octave_call "norm" 1 [x, 1];
6.0
```

Instead of a function name, you can also specify the function to be called using a special kind of Octave object, a function value. These are returned, e.g., by Octave's `str2func` and `inline` builtins. For your convenience, pure-octave provides a frontend to these builtins, the `octave_func` function, which lets you specify an Octave function in one of two ways:

octave_func name
: Returns the Octave function with the given name. This works like Octave's `str2func` builtin.

octave_func expr
: Returns an "inline" function, where `expr` is an Octave expression (as a string) describing the function value. This works like Octave's `inline` builtin. Instead of just an Octave expression, you can also specify a tuple or a list consisting of the inline expression and the parameter names. (Otherwise the parameters are determined automatically, see the description of the `inline` function in the Octave manual for details.)

Note that inline functions allow you to call stuff that is not an Octave function and hence cannot be specified directly in `octave_call`, such as an operator. Examples:

```
> let eig = octave_func "eig";
> let mul = octave_func "x*y";
> octave_call eig 1 (octave_call mul 1 (x,x));
{0.138593383654928;28.8614066163451}
> let add = octave_func ("x+y","x","y");
> octave_call add 1 (x,x);
{2.0,4.0;6.0,8.0}
```

17.6 Data Conversions

As shown above, the `octave_set`, `octave_get` and `octave_call` functions convert Pure data to corresponding Octave values and vice versa. Octave scalars and matrices of boolean, integer, double, complex and character data are all supported by this interface, and are mapped to the corresponding Pure data types in a straightforward manner (scalars to scalars, matrices to matrices and strings to strings). Note that in any case these conversions create *copies* of the data, so modifying, say, an Octave matrix received via `octave_get` in Pure only affects the Pure copy of the matrix and leaves the original Octave matrix unchanged.

Any other kind of Octave object (including Octave function objects, see Direct Function Calls) is just passed through as is, in the form of a cooked pointer to an Octave value which frees itself when garbage-collected. You can check for such objects with the `octave_valuep` predicate:

octave_valuep x
: Check for Octave value pointers.

```
> let eig = octave_func "eig";
> eig; octave_valuep eig;
#<pointer 0x230dba0>
1
```

Such Octave value pointers can be used freely whereever an Octave value is needed (i.e., in invocations of `octave_set` and `octave_call`).

You should also note the following:

- Octave's cell arrays and structures are not supported at this time, so they will show up as opaque Octave value pointers in Pure land. This allows these objects to be passed around freely, but you can't inspect or modify them in Pure.
- Scalars and 1x1 matrices are indistinguishable in Octave, thus any 1x1 matrix will be returned as a scalar from Octave to Pure.
- All types of boolean and integer matrices are returned from Octave to Pure as (machine) integer matrices. When converted back to Octave, these all become Octave `int32` matrices, but you can easily convert them to boolean or other types of matrices in Octave as needed. For instance:

```
> octave_set "a" {1,2;3,4};
{1,2;3,4}
> octave_eval "global a ans";
0
> octave_eval "eig(a)";
error: eig: wrong type argument 'int32 matrix'
1
> octave_eval "eig(double(a))";
ans =

  -0.37228
   5.37228

0
> octave_eval "a>2";
ans =

   0   0
   1   1

0
> octave_get "ans";
{0,0;1,1}
```

- Octave strings are mapped to Pure strings, and character matrices with more than one row are mapped to (symbolic) column vectors of Pure strings. Example:

```
> octave_set "a" "Hello, world!";
"Hello, world!"
> octave_eval "a";
a = Hello, world!
0
> octave_eval "[a;'abc']";
ans =

Hello, world!
abc

0
> octave_get "ans";
{"Hello, world!";"abc          "}
```

17.7 Calling Back Into Pure

The embedded Octave interpreter provides one special builtin, the `pure_call` function which can be used to call any function defined in the executing Pure script from Octave. For instance:

```
> even m::matrix = {~(int x mod 2) | x=m};
> octave_eval "pure_call('even', 1:12)";
ans =

  0  1  0  1  0  1  0  1  0  1  0  1

0
```

Here's the description of the `pure_call` function, as it is printed with Octave's `help` command:

```
'pure_call' is a built-in function

  RES = pure_call(NAME, ARG, ...)
  [RES, ...] = pure_call(NAME, ARG, ...)

  Execute the Pure function named NAME (a string) with the given arguments.
  Arguments and result types may be scalars and matrices of boolean, integer,
  double, complex and character data. The Pure function may return multiple
  results as a tuple. Example: pure_call('succ', 99) => 100.
```

17.8 Caveats and Notes

Directly embedding Octave in Pure programs is convenient because it allows easy exchange of data between Pure and Octave, and you can also call Octave functions directly from Pure and vice versa. However, it also comes at a cost. A default build of Octave pulls in quite a few dependencies of its own which might conflict with other modules loaded in a Pure script. Specifically, we have found that older Octave versions may give problems with third-party graphics libraries such as VTK, if used in the same program as Octave. (These seem to be fixed in the latest Octave and VTK versions, however.)

Chapter 18

Pure-Rational - Rational number library for the Pure programming language

Version 0.1, June 26, 2012

Rob Hubbard
Albert Graef <Dr.Graef@t-online.de>
Jiri Spitz <jiri.spitz@bluetone.cz>

This package provides a Pure port of Q+Q, Rob Hubbard's rational number library for the Q programming language. The port was done by Jiri Spitz. It contains `rational.pure`, a collection of utility functions for rational numbers, and `rat_interval.pure`, a module for doing interval arithmetic needed by `rational.pure`. These modules are designed to work with the `math.pure` module (part of the standard Pure library), which contains the definition of Pure's rational type and implements the basic rational arithmetic.

This document is an edited version of Rob's original Q+Q manual available from the Q website, slightly adjusted to account for the Pure specifics of the implementation. In particular, note that the operations provided by `rational.pure` and `rat_interval.pure` live in their own `rational` and `interval` namespaces, respectively, so if you want to get unqualified access to the symbols of these modules (as the examples in this manual assume) then you'll have to import the modules as follows:

```
using rational, rat_interval;
using namespace rational, interval;
```

Also note that `rational` always pulls in the `math` module, so you don't have to import the latter explicitly if you are using `rational`.

Another minor difference to the Q version of this module is that rational results always have Pure bigints as their numerators and denominators, hence the `L` suffix in the printed results.

Also, unary minus binds weaker in Pure than the rational division operator, so a negative rational number will be printed as, e.g., `(-1L)%2L`, which looks a bit funny but is correct since Pure rationals always carry their sign in the numerator.

18.1 Copying

Copyright (c) 2006 - 2010 by Rob Hubbard.
Copyright (c) 2006 - 2010 by Albert Graef <Dr.Graef@t-online.de>.
Copyright (c) 2010 by Jiri Spitz <jiri.spitz@bluetone.cz>.

Pure-rational is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

18.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-rational-0.1.tar.gz.

Then run `make install` (as root) to install pure-rational in the Pure library directory. This requires GNU make, and of course you need to have Pure installed.

`make install` tries to guess your Pure installation directory. If it gets this wrong, you can install using `make install prefix=/usr` which sets the installation prefix. Please see the Makefile for details.

18.3 Introduction

18.3.1 The Rational Module

This module provides additional operations on the rational number type provided by the `math.pure` module in the standard library. The module is compatible with Pure version 0.43 (onwards).

18.3.2 The Files and the Default Prelude

The implementation of the rational type and associated utilities is distributed across various files.

math.pure and Other Files

The file `math.pure` defines the type, its constructors and 'deconstructors' and basic arithmetical and mathematical operators and functions. This is part of the standard Pure library. A few definitions associated with rationals are also defined in other standard library modules. In particular, the type tests are contained in `primitives.pure`.

It is also possible to create rational complex numbers (in addition to double complex numbers and integral or Gaussian complex numbers). That is, rationals play nicely with the complex number constructors provided in the `math.pure` module. This is discussed further in Rational Complex Numbers.

rational.pure

Additional 'rational utilities', not included in the `math.pure` module, are defined in `rational.pure`. The functions include further arithmetical and mathematical operators and functions, continued fraction support, approximation routines and string formatting and evaluation.

The rational utilities include some 'rational complex number' functions.

rat_interval.pure

Amongst the rational utilities are some functions that return a rational interval. The file `rat_interval.pure` is a partial implementation of interval arithmetic. Intervals are discussed further in Intervals.

18.3.3 Notation

Throughout this document, the parameters q, q0, q1, ... usually denote rationals ($\in \mathbf{Q}$), parameters z, ... usually denote integers ($\in \mathbf{Z}$), r, ... usually denote real numbers ($\in \mathbf{R}$), c, ... usually denote complex numbers ($\in \mathbf{C}$), n, ... usually denote parameters of any numeric type, v, ... usually denote parameters of any interval type, and x, ... usually denote parameters of any type.

The reals are not just the doubles, but include rationals and integers. The term 'rational' usually refers to a rational number $\in \mathbf{Q} \supset \mathbf{Z}$, or an expression of type rational or integer.

18.4 The Rational Type

18.4.1 Constructors

Rationals are constructed with the `%` exact division operator, and other kinds of numbers can be converted to rationals with the `rational` function. These are both defined in math.pure.

n1 **`%`** n2
: is the exact division operator, which may be used as a constructor (for integers n1 and n2). This is described in More on Division.

`rational` x
: converts the given number x to a rational.

Example 1 Constructing a fraction:

```
> 44%14;
22L%7L
```

Example 2 Converting from an integer:

```
> rational 3;
3L%1L
```

18.4.2 'Deconstructors'

A rational number is in simplest form if the numerator and denominator are coprime (i.e. do not have a factor in common) and the denominator is positive (and, specifically, non-zero). Sometimes the term 'irreducible' is used for a rational in simplest form. This is a property of the representation of the rational number and not of the number itself.

`num` q
: given a rational or integer q, returns the '(signed) simplest numerator', i.e. the numerator of the normalised form of q.

`den` q
: given a rational or integer q, returns the '(positive) simplest denominator', i.e. the denominator of the normalised form of q.

`rational::`**`num_den`** q
: given a rational or integer q, returns a pair (n, d) containing the (signed) simplest numerator n and the (positive) simplest denominator d. This is the inverse (up to equivalence) of rational as defined on integer pairs (see Constructors).

Example 3 Using num_den to obtain a representation in simplest form:

```
> let q = (44%(-14));
> num q;
-22L
> den q;
7L
```

```
> num_den q;
-22L,7L
> num_den 3;
3L,1L
> num_den (-3);
-3L,1L
```

Together, num and den are a pair of 'decomposition' operators, and num_den is also a decomposition operator. There are others (see Decomposition). The integer and fraction function (see Integer and Fraction Parts) may be used in conjunction with num_den_gauss to decompose a rational into integer, numerator and denominator parts.

18.4.3 Type and Value Tests

The functions rationalp and ratvalp and other rational variants are new for rationals and the standard functions exactp and inexactp are extended for rationals.

A value is 'exact', or of an exact type, if it is of a type that is able to represent the values returned by arithmetical operations exactly; in a sense, it is 'closed' under arithmetical operations. Otherwise, a value is 'inexact'. Inexact types are able to store some values only approximately.

The doubles are not an exact type. The results of some operations on some values that are stored exactly, can't be stored exactly. (Furthermore, doubles are intended to represent real numbers; no irrational number ($\in \mathbf{R} \setminus \mathbf{Q}$) can be stored exactly as a double; even some rational ($\in \mathbf{Q}$) numbers are not stored exactly.)

The rationals are an exact type. All rational numbers (subject to available resources, of course) are stored exactly. The results of the arithmetical operations on rationals are rationals represented exactly. Beware that the standard intvalp and ratvalp may return 1 even if the value is of double type. However, these functions may be combined with exactp.

exactp x
: returns whether x has an exact value.

inexactp x
: returns whether x has an inexact value.

rationalp x
: returns whether x is of rational type.

ratvalp x
: returns whether x has a rational value.

Example 4 Rational value tests:

```
> let l = [9, 9%1, 9%2, 4.5, sqrt 2, 1+i, inf, nan];
> map exactp l;
[1,1,1,0,0,1,0,0]
> map inexactp l;
[0,0,0,1,1,0,1,1]
```

```
> map rationalp l;
[0,1,1,0,0,0,0,0]
> map ratvalp l;
[1,1,1,1,1,0,0,0]
> map (\x -> (exactp x && ratvalp x)) l; // "has exact rational value"
[1,1,1,0,0,0,0,0]
> map intvalp l; // for comparison
[1,1,0,0,0,0,0,0]
> map (\x -> (exactp x && intvalp x)) l; // "has exact integer value"
[1,1,0,0,0,0,0,0]
```

See Rational Complex Numbers for details about rational complex numbers, and Rational Complex Type and Value Tests for details of their type and value tests.

18.5 Arithmetic

18.5.1 Operators

The standard arithmetic operators (+), (−) and (*) are overloaded to have at least one rational operand. If both operands are rational then the result is rational. If one operand is integer, then the result is rational. If one operand is double, then the result is double.

The operators (/) and (%) are overloaded for division on at least one rational operand. The value returned by (/) is always `inexact` (in the sense of Type and Value Tests). The value returned by (%) is `exact` (if it exists).

The standard function `pow` is overloaded to have a rational left operand. If `pow` is passed integer operands where the right operand is negative, then a rational is returned. The right operand should be an integer; negative values are permitted (because $q^{-z} = 1/q^z$). It is not overloaded to also have a rational right operand because such values are not generally rational (e.g. $q^{1/n} = \sqrt[n]{q}$).

The standard arithmetic operator (^) is also overloaded, but produces a double value (as always).

Example 5 Arithmetic:

```
> 5%7 + 2%3;
29L%21L
> str_mixed ans;
"1L+8L/21L"
> 1 + 2%3;
5L%3L
> ans + 1.0;
2.66666666666667
> 3%8 - 1%3;
1L%24L
> (11%10) ^ 3;
1.331
> pow (11%10) 3;
```

```
1331L%1000L
> pow 3 5;
243L
> pow 3 (-5);
1L%243L
```

(See the function `str_mixed`.)

Beware that (/) on integers will not produce a rational result.

Example 6 Division:

```
> 44/14;
3.14285714285714
> 44%14;
22L%7L
> str_mixed ans;
"3L+1L/7L"
```

(See the function `str_mixed`.)

18.5.2 More on Division

There is a rational-aware divide operator on the numeric types:

n1 **%** n2
: returns the quotient ($\in \mathbf{Q}$) of n1 and n2. If n1 and n2 are rational or integer then the result is rational. This operator has the precedence of division (/).

Example 7 Using % like a constructor:

```
> 44 % 14;
22L%7L
> 2 + 3%8; // "2 3/8"
19L%8L
> str_mixed ans;
"2L+3L/8L"
```

(See the function `str_mixed`.)

`rational::`**`reciprocal`** n
: returns the reciprocal of n: 1/n.

Example 8 Reciprocal:

```
> reciprocal (22%7);
7L%22L
```

The following division functions are parameterised by a rounding mode `roundfun`. The available rounding modes are described in Rounding to Integer.

`rational::`**`divide`** roundfun n d
: for rationals n and d returns a pair (q, r) of 'quotient' and 'remainder' where q is an

integer and r is a rational such that |r| < |d| (or better) and n = q * d + r. Further conditions may hold, depending on the chosen rounding mode `roundfun` (see Rounding to Integer). If `roundfun` = `floor` then $0 \leq r < d$. If `roundfun` = `ceil` then $-d < r \leq 0$. If `roundfun` = `trunc` then |r| < |d| and sgn r ∈ {0, sgn d}. If `roundfun` = `round`, `roundfun` = `round_zero_bias` or `roundfun` = `round_unbiased` then $|r| \leq d/2$.

`rational::`**`quotient`** roundfun nN d
: returns just the quotient as produced by `divide` roundfun n d.

`rational::`**`modulus`** roundfun n d
: returns just the remainder as produced by `divide` roundfun n d.

q1 **`div`** q2
: (overload of the built-in div) q1 and q2 may be rational or integer. Returns an integer.

q1 **`mod`** q2
: (overload of the built-in mod) q1 and q2 may be rational or integer. Returns a rational. If q = q1 div q2 and r = q1 mod q2 then q1 = q * q2 + q, q ∈ **Z**, |r| < |q2| and sgn r ∈ {0, sgn q2}.

18.5.3 Relations — Equality and Inequality Tests

The standard arithmetic operators (==), (~=), (<), (<=), (>), (>=) are overloaded to have at least one rational operand. The other operand may be rational, integer or double.

Example 9 Inequality:

```
> 3%8 < 1%3;
0
```

18.5.4 Comparison Function

`rational::`**`cmp`** n1 n2
: is the 'comparison' (or 'compare') function, and returns sgn (n1 − n2); that is, it returns −1 if n1 < n2, 0 if n1 = n2, and +1 if n1 > n2.

Example 10 Compare:

```
> cmp (3%8) (1%3);
1
```

18.6 Mathematical Functions

Most mathematical functions, including the elementary functions (sin, $\sin^{-1}$, sinh, $\sinh^{-1}$, cos, ... , exp, ln, ...), are not closed on the set of rational numbers. That is, most mathematical functions do not yield a rational number in general when applied to a rational number.

Therefore the elementary functions are not defined for rationals. To apply these functions, first apply a cast to double, or compose the function with a cast.

18.6.1 Absolute Value and Sign

The standard `abs` and `sgn` functions are overloaded for rationals.

abs q
: returns absolute value, or magnitude, |q| of q; abs q = |q| = q × sgn q (see below).

sgn q
: returns the sign of q as an integer; returns −1 if q < 0, 0 if q = 0, +1 if q > 0.

Together, these functions satisfy the property ∀q • (sgn q) * (abs q) = q (i.e. ∀q • (sgn q) * |q| = q). Thus these provide a pair of 'decomposition' operators; there are others (see Decomposition).

18.6.2 Greatest Common Divisor (GCD) and Least Common Multiple (LCM)

The standard functions `gcd` and `lcm` are overloaded for rationals, and mixtures of integer and rational.

gcd n1 n2
: The GCD is also known as the Highest Common Factor (HCF). The GCD of rationals q1 and q2 is the largest (therefore positive) rational f such that f divides into both q1 and q2 exactly, i.e. an integral number of times. This is not defined for n1 and n2 both zero. For integral q1 and q2, this definition coincides with the usual definition of GCD for integers.

Example 11 With two rationals:

```
> let a = 7%12;
> let b = 21%32;
> let f = gcd a b;
> f;
7L%96L
> a % f;
8L%1L
> b % f;
9L%1L
```

Example 12 With a rational and an integer:

```
> let f = gcd (6%5) 4;
> f;
2L%5L
> (6%5) % f;
3L%1L
> 4 % f;
10L%1L
```

Example 13 With integral rationals and with integers:

```
> gcd (rational 18) (rational 24);
6L%1L
> gcd 18 24;
6
```

Example 14 The behaviour with negative numbers:

```
> gcd (rational (-18)) (rational 24);
6L%1L
> gcd (rational 18) (rational (-24));
6L%1L
> gcd (rational (-18)) (rational (-24));
6L%1L
```

lcm n1 n2

The LCM of rationals q1 and q2 is the smallest positive rational m such that both q1 and q2 divide m exactly. This is not defined for n1 and n2 both zero. For integral q1 and q2, this definition coincides with the usual definition of LCM for integers.

Example 15 With two rationals:

```
> let a = 7%12;
> let bB = 21%32;
> let m = lcm a b;
> m;
21L%4L
> m % a;
9L%1L
> m % b;
8L%1L
```

Example 16 With a rational and an integer:

```
> let m = lcm (6%5) 4;
> m;
12L%1L
> m % (6%5);
10L%1L
```

Example 17 The behaviour with negative numbers:

```
> lcm (rational (-18)) (rational 24);
72L%1L
> lcm (rational 18) (rational (-24));
72L%1L
> lcm (rational (-18)) (rational (-24));
72L%1L
```

Together, the GCD and LCM have the following property when applied to two numbers: (gcd q1 q2) * (lcm q1 q2) = |q1 * q2|.

18.6.3 Extrema (Minima and Maxima)

The standard `min` and `max` functions work with rational values.

Example 18 Maximum:

```
> max (3%8) (1%3);
3L%8L
```

18.7 Special Rational Functions

18.7.1 Complexity

The 'complexity' (or 'complicatedness') of a rational is a measure of the greatness of its simplest (positive) denominator.

The complexity of a number is not itself made available, but various functions and operators are provided to allow complexities to be compared. Generally, it does not make sense to operate directly on complexity values.

The complexity functions in this section may be applied to integers (the least complex), rationals, or reals (doubles; the most complex).

Functions concerning 'complexity' are named with 'cplx', whereas functions concerning 'complex numbers' (see Rational Complex Numbers) are named with 'comp'.

Complexity Relations

n1 `rational::`**`eq_cplx`** n2
: "[is] equally complex [to]" — returns 1 if n1 and n2 are equally complex; returns 0 otherwise. Equal complexity is not the same a equality; n1 and n2 are equally complex if their simplest denominators are equal. Equal complexity forms an equivalence relation on rationals.

Example 19 Complexity equality test:

```
> (1%3) eq_cplx (100%3);
1
> (1%4) eq_cplx (1%5);
0
> (3%3) eq_cplx (1%3); // LHS is not in simplest form
0
```

n1 `rational::`**`not_eq_cplx`** n2
: "not equally complex" — returns 0 if n1 and n2 are equally complex; returns 1 otherwise.

n1 `rational::`**`less_cplx`** n2
: "[is] less complex [than]" (or "simpler") — returns 1 if n1 is strictly less complex than n2; returns 0 otherwise. This forms a partial strict ordering on rationals.

Example 20 Complexity inequality test:

```
> (1%3) less_cplx (100%3);
0
> (1%4) less_cplx (1%5);
1
> (3%3) less_cplx (1%3); // LHS is not in simplest form
1
```

n1 `rational::`**`less_eq_cplx`** n2
: "less or equally complex" (or "not more complex") — returns 1 if n1 is less complex than or equally complex to n2; returns 0 otherwise. This forms a partial non-strict ordering on rationals.

n1 `rational::`**`more_cplx`** n2
: "[is] more complex [than]" — returns 1 if n1 is strictly more complex than n2; returns 0 otherwise. This forms a partial strict ordering on rationals.

n1 `rational::`**`more_eq_cplx`** n2
: "more or equally complex" (or "not less complex") — returns 1 if n1 is more complex than or equally complex to n2; returns 0 otherwise. This forms a partial non-strict ordering on rationals.

Complexity Comparison Function

`rational::`**`cmp_complexity`** n1 n2
: is the 'complexity comparison' function, and returns the sign of the difference in complexity; that is, it returns -1 if n1 is less complex than n2, 0 if n1 and n2 are equally complex (but not necessarily equal), and +1 if n1 is more complex than n2.

Example 21 Complexity comparison:

```
> cmp_complexity (1%3) (100%3);
0
> cmp_complexity (1%4) (1%5);
-1
> cmp_complexity (3%3) (1%3); // LHS is not in simplest form
-1
```

Complexity Extrema

`rational::`**`least_cplx`** n1 n2
: returns the least complex of n1 and n2; if they're equally complex, n1 is returned.

Example 22 Complexity selection:

```
> least_cplx (100%3) (1%3);
100L%3L
> least_cplx (1%5) (1%4);
1L%4L
> least_cplx (1%3) (3%3); // second argument not in simplest form
1L%1L
```

`rational::`**`most_cplx`** n1 n2
: returns the most complex of n1 and n2; if they're equally complex, n1 is returned.

Other Complexity Functions

`rational::`**`complexity_rel`** n1 op n2
: returns "complexity-of n1" compared by operator op to the "complexity-of n2". This is equivalent to prefix complexity rel op n1 n2 (below), but is the more readable form.

Example 23 Complexity relations:

```
> complexity_rel (1%3) (==) (100%3);
1
> complexity_rel (1%4) (<=) (1%5);
1
> complexity_rel (1%4) (>) (1%5);
0
```

`rational::`**`_complexity_rel`** op n1 n2
: returns the same as complexity_rel n1 op n2, but this form is more convenient for currying.

18.7.2 Mediants and Farey Sequences

`rational::`**`mediant`** q1 q2
: returns the canonical mediant of the rationals q1 and q2, a form of (nonarithmetic) average on rationals. The mediant of the representations n1/d1 = q1 and n2/d2 = q2, where d1 and d2 must be positive, is defined as (n1 + n2)/(d1 + d2). A mediant of the rationals q1 and q2 is a mediant of some representation of each of q1 and q2. That is, the mediant is dependent upon the representations and therefore is not well-defined as a function on pairs of rationals. The canonical mediant always assumes the simplest representation, and therefore is well-defined as a function on pairs of rationals.

  By the phrase "the mediant" (as opposed to just "a mediant") we always mean "the canonical mediant".

  If q1 < q2, then any mediant q is always such that q1 < q < q2.

  The (canonical) mediant has some special properties. If q1 and q2 are integers, then the mediant is the arithmetic mean. If q1 and q2 are unit fractions (reciprocals of integers), then the mediant is the harmonic mean. The mediant of q and 1/q is ± 1, (which

happens to be a geometric mean with the correct sign, although this is a somewhat uninteresting and degenerate case).

Example 24 Mediants:

```
> mediant (1%4) (3%10);
2L%7L
> mediant 3 7; // both integers
5L%1L
> mediant 3 8; // both integers again
11L%2L
> mediant (1%3) (1%7); // both unit fractions
1L%5L
> mediant (1%3) (1%8); // both unit fractions again
2L%11L
> mediant (-10) (-1%10);
(-1L)%1L
```

`rational::`**`farey`** k
: for an integer k, farey returns the ordered list containing the order-k Farey sequence, which is the ordered list of all rational numbers between 0 and 1 inclusive with (simplest) denominator at most k.

Example 25 A Farey sequence:

```
> map str_mixed (farey 6);
["0L","1L/6L","1L/5L","1L/4L","1L/3L","2L/5L","1L/2L","3L/5L","2L/3L",
"3L/4L","4L/5L","5L/6L","1L"]
```

(See the function `str_mixed`.)

Farey sequences and mediants are closely related. Three rationals q1 < q2 < q3 are consecutive members of a Farey sequence if and only if q2 is the mediant of q1 and q3. If rationals q1 = n1/d1 < q2 = n2/d2 are consecutive members of a Farey sequence, then n2d1 − n1d2 = 1.

18.7.3 Rational Type Simplification

`rational::`**`rat_simplify`** q
: returns q with rationals simplified to integers, if possible.

Example 26 Rational type simplification:

```
> let l = [9, 9%1, 9%2, 4.5, 9%1+i, 9%2+i]; l;
[9,9L%1L,9L%2L,4.5,9L%1L+:1,9L%2L+:1]
> map rat_simplify l;
[9,9,9L%2L,4.5,9+:1,9L%2L+:1]
```

See Rational Complex Numbers for details about rational complex numbers, and Rational Complex Type Simplification for details of their type simplification.

18.8 Q -> Z — Rounding

18.8.1 Rounding to Integer

Some of these are new functions, and some are overloads of standard functions. The behaviour of the overloads is consistent with that of the standard functions.

floor q
: (overload of standard function) returns q rounded downwards, i.e. towards -1, to an integer, usually denoted bQc.

ceil q
: (overload of standard function) returns q rounded upwards, i.e. towards +1, to an integer, usually denoted dQe.

trunc q
: (overload of standard function) returns q truncated, i.e. rounded towards 0, to an integer.

round q
: (overload of standard function) returns q 'rounded off', i.e. rounded to the nearest integer, with 'half-integers' (values that are an integer plus a half) rounded away from zero.

rational::**round_zero_bias** q
: (new function) returns q 'rounded off', i.e. rounded to the nearest integer, but with 'half-integers' rounded towards zero.

rational::**round_unbiased** q
: (new function) returns q rounded to the nearest integer, with 'half-integers' rounded to the nearest even integer.

Example 27 Illustration of the different rounding modes:

```
> let l = iterwhile (<= 3) (+(1%2)) (- rational 3);
> map double l; // (just to show the values in a familiar format)
[-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0]
> map floor l;
[-3L,-3L,-2L,-2L,-1L,-1L,0L,0L,1L,1L,2L,2L,3L]
> map ceil l;
[-3L,-2L,-2L,-1L,-1L,0L,0L,1L,1L,2L,2L,3L,3L]
> map trunc l;
[-3L,-2L,-2L,-1L,-1L,0L,0L,0L,1L,1L,2L,2L,3L]
> map round l;
[-3L,-3L,-2L,-2L,-1L,-1L,0L,1L,1L,2L,2L,3L,3L]
> map round_zero_bias l;
[-3L,-2L,-2L,-1L,-1L,0L,0L,0L,1L,1L,2L,2L,3L]
> map round_unbiased l;
[-3L,-2L,-2L,-2L,-1L,0L,0L,0L,1L,2L,2L,2L,3L]
```

(See the function `double`.)

18.8.2 Integer and Fraction Parts

`rational::`**`integer_and_fraction`** roundfun q
: returns a pair (z, f) where z is the 'integer part' as an integer, f is the 'fraction part' as a rational, where the rounding operations are performed using rounding mode `roundfun` (see Rounding to Integer).

Example 28 Integer and fraction parts with the different rounding modes:

```
> let nc = -22%7;
> integer_and_fraction floor nc;
-4L,6L%7L
> integer_and_fraction trunc nc;
-3L,(-1L)%7L
> integer_and_fraction round nc;
-3L,(-1L)%7L
```

It is always the case that z and f have the property that q = z + f. However, the remaining properties depend upon the choice of `roundfun`. Thus this provides a 'decomposition' operator; there are others (see Decomposition). If `roundfun` = `floor` then $0 \leq f < 1$. If `roundfun` = `ceil` then $-1 < f \leq 0$. If `roundfun` = `trunc` then $|f| < 1$ and sgn f $\in \{0, \text{sgn } q\}$. If `roundfun` = `round`, `roundfun` = `round_zero_bias` or `roundfun` = `round_unbiased` then $|f| \leq 1/2$.

`rational::`**`fraction`** roundfun q
: returns just the 'fraction part' as a rational, where the rounding operations are performed using `roundfun`. The corresponding function 'integer' is not provided, as integer `roundfun` q would be just `roundfun` q. The integer and fraction function (probably with `trunc` or `floor` rounding mode) may be used in conjunction with `num_den` (see 'Deconstructors') to decompose a rational into integer, numerator and denominator parts.

`int` q
: overloads the built-in int and returns the 'integer part' of q consistent with the built-in.

`frac` q
: overloads the built-in frac and returns the 'fraction part' of q consistent with the built-in.

Example 29 Standard integer and fraction parts:

```
> let nc = -22%7;
> int nc;
-3
> frac nc;
(-1L)%7L
```

18.9 Rounding to Multiples

`rational::`**`round_to_multiple`** roundfun multOf q
: returns q rounded to an integer multiple of a non-zero value multOf, using `roundfun`

as the rounding mode (see Rounding to Integer). Note that it is the multiple that is rounded in the prescribed way, and not the final result, which may make a difference in the case that multOf is negative. If that is not the desired behaviour, pass this function the absolute value of multOf rather than multOf. Similar comments apply to the following functions.

`rational::`**`floor_multiple`** multOf q
: returns q rounded to a downwards integer multiple of multOf.

`rational::`**`ceil_multiple`** multOf q
: returns q rounded to an upwards integer multiple of multOf.

`rational::`**`trunc_multiple`** multOf q
: returns q rounded towards zero to an integer multiple of multOf.

`rational::`**`round_multiple`** multOf q
: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded away from 0.

`rational::`**`round_multiple_zero_bias`** multOf q
: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded towards 0.

`rational::`**`round_multiple_unbiased`** multOf q
: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded to an even multiple.

Example 30 Round to multiple:

```
> let l = [34.9, 35, 35%1, 35.0, 35.1];
> map double l; // (just to show the values in a familiar format)
[34.9,35.0,35.0,35.0,35.1]
> map (floor_multiple 10) l;
[30.0,30L,30L,30.0,30.0]
> map (ceil_multiple 10) l;
[40.0,40L,40L,40.0,40.0]
> map (trunc_multiple 10) l;
[30.0,30L,30L,30.0,30.0]
> map (round_multiple 10) l;
[30.0,40L,40L,40.0,40.0]
> map (round_multiple_zero_bias 10) l;
[30.0,30L,30L,30.0,40.0]
> map (round_multiple_unbiased 10) l;
[30.0,40L,40L,40.0,40.0]
```

(See the function `double`.)

The round multiple functions may be used to find a fixed denominator approximation of a number. (The simplest denominator may actually be a proper factor of the chosen value.) To approximate for a bounded (rather than particular) denominator, use rational approx max den instead (see Best Approximation with Bounded Denominator).

Example 31 Finding the nearest q = n/d value to $1/e \approx 0.368$ where d = 1000 (actually, where d | 1000):

```
> let co_E = exp (-1);
co_E;
0.367879441171442
> round_multiple (1%1000) (rational co_E);
46L%125L
> 1000 * ans;
368L%1L
```

Example 32 Finding the nearest q = n/d value to $1/\phi \approx 0.618$ where d = 3^5 = 243 (actually, where d | 243):

```
> let co_Phi = (sqrt 5 - 1) / 2;
> round_multiple (1%243) (rational co_Phi);
50L%81L
```

Other methods for obtaining a rational approximation of a number are described in R -> Q — Approximation.

18.10 Q -> R — Conversion / Casting

double q
: (overload of built-in) returns a double having a value as close as possible to q. (Overflow, underflow and loss of accuracy are potential problems. rationals that are too absolutely large or too absolutely small may overflow or underflow; some rationals can not be represented exactly as a double.)

18.11 R -> Q — Approximation

This section describes functions that approximate a number (usually a double) by a rational. See Rounding to Multiples for approximation of a number by a rational with a fixed denominator. See Numeral String -> Q — Approximation for approximation by a rational of a string representation of a real number.

18.11.1 Intervals

Some of the approximation functions return an **interval**. The file `rat_interval.pure` is a basic implementation of interval arithmetic, and is not included in the default prelude. It is not intended to provide a complete implementation of interval arithmetic. The notions of 'open' and 'closed' intervals are not distinguished. Infinite and half-infinite intervals are not specifically provided. Some operations and functions may be missing. The most likely functions to be used are simply the 'deconstructors'; see Interval Constructors and 'Deconstructors'.

Interval Constructors and 'Deconstructors'

Intervals are constructed with the function interval.

`interval::`**`interval`** (n1, n2)
: given a pair of numbers (z1 <= z2), this returns the interval z1..z2. This is the inverse of `lo_up`.

Example 33 Constructing an interval:

```
> let v = interval (3, 8);
> v;
interval::Ivl 3 8
```

`interval::`**`lower`** v
: returns the infimum (roughly, minimum) of v.

`interval::`**`upper`** v
: returns the supremum (roughly, maximum) of v.

`interval::`**`lo_up`** v
: returns a pair (l, u) containing the lower l and upper u extrema of the interval v. This is the inverse of interval as defined on number pairs.

Example 34 Deconstructing an interval:

```
> lower v;
3
> upper v;
8
> lo_up v;
3,8
```

Interval Type Tests

`exactp` v
: returns whether an interval v has exact extrema.

`inexactp` v
: returns whether an interval v has an inexact extremum.

`interval::`**`intervalp`** x
: returns whether x is of type interval.

`interval::`**`interval_valp`** x
: returns whether x has an interval value.

`interval::`**`ratinterval_valp`** x
: returns whether x has an interval value with rational extrema.

`interval::`**`intinterval_valp`** x
: returns whether x has an interval value with integral extrema.

Example 35 Interval value tests:

```
> let l = [interval(0,1), interval(0,1%1), interval(0,3%2), interval(0,1.5)];
> map exactp l;
[1,1,1,0]
> map inexactp l;
[0,0,0,1]
> map intervalp l;
[1,1,1,1]
> map interval_valp l;
[1,1,1,1]
> map ratinterval_valp l;
[1,1,1,1]
> map intinterval_valp l;
[1,1,0,0]
```

Interval Arithmetic Operators and Relations

The standard arithmetic operators (+), (−), (*), (/) and (%) are overloaded for intervals. The divide operators (/) and (%) do not produce a result if the right operand is an interval containing 0. **Example 36** Some intervals:

```
> let a = interval (11, 19);
> let b = interval (16, 24);
> let c = interval (21, 29);
> let d = interval (23, 27);
```

Example 37 Interval arithmetic:

```
> let p = interval (0, 1);
> let s = interval (-1, 1);
> a + b;
interval::Ivl 27 43
> a - b;
interval::Ivl (-13) 3
> a * b;
interval::Ivl 176 456
> p * 2;
interval::Ivl 0 2
> (-2) * p;
interval::Ivl (-2) 0
> -c;
interval::Ivl (-29) (-21)
> s * a;
interval::Ivl (-19) 19
> a % 2;
interval::Ivl (11L%2L) (19L%2L)
> a / 2;
interval::Ivl 5.5 9.5
> reciprocal a;
interval::Ivl (1L%19L) (1L%11L)
```

```
> 2 % a;
interval::Ivl (2L%19L) (2L%11L)
> a % b;
interval::Ivl (11L%24L) (19L%16L)
> a % a; // notice that the intervals are mutually independent here
interval::Ivl (11L%19L) (19L%11L)
```

There are also some relations defined for intervals. The standard relations (==) and (~=) are overloaded.

However, rather than overloading (<), (<=), (>), (>=), which could be used for either ordering or containment with some ambiguity, the module defines `(before)`, `(within)`, and so on. 'Strictness' refers to the properties at the end-points.

v1 `interval::`**`before`** v2
: returns whether v1 is entirely before v2.

v1 `interval::`**`strictly_before`** v2
: returns whether v1 is strictly entirely before v2.

v1 `interval::`**`after`** v2
: returns whether v1 is entirely after v2.

v1 `interval::`**`strictly_after`** v2
: returns whether v1 is strictly entirely after v2.

v1 `interval::`**`within`** v2
: returns whether v1 is entirely within v2; i.e. whether v1 is subinterval of v2.

v1 `interval::`**`strictly_within`** v2
: returns whether v1 is strictly entirely within v2; i.e. whether v1 is proper subinterval of v2.

v1 `interval::`**`without`** v2
: returns whether v1 entirely contains v2; i.e. whether v1 is superinterval of v2. 'Without' is used in the sense of outside or around.

v1 `interval::`**`strictly_without`** v2
: returns whether v1 strictly entirely contains v2; i.e. whether v1 is proper superinterval of v2.

v1 `interval::`**`disjoint`** v2
: returns whether v1 and v2 are entirely disjoint.

v `interval::`**`strictly_disjoint`** v2
: returns whether v1 and v2 are entirely strictly disjoint.

Example 38 Interval relations:

```
> a == b;
0
> a == a;
1
> a before b;
```

```
0
> a before c;
1
> c before a;
0
> a disjoint b;
0
> c disjoint a;
1
> a within b;
0
> a within c;
0
> d within c;
1
> c within d;
0
> a strictly_within a;
0
> a within a;
1
```

(The symbols a through d were defined in Example 36.)

These may also be used with a simple (real) value, and in particular to test membership.

Example 39 Membership:

```
> 10 within a;
0
> 11 within a;
1
> 11.0 within a;
1
> 12 within a;
1
> 12.0 within a;
1
> 10 strictly_within a;
0
> 11 strictly_within a;
0
> (11%1) strictly_within a;
0
> 12 strictly_within a;
1
> (12%1) strictly_within a;
1
```

(The symbol a was defined in Example 36.)

Interval Maths

Some standard functions are overloaded for intervals; some new functions are provided.

abs v
: returns the interval representing the range of (x) as x varies over v.

Example 40 Absolute interval:

```
> abs (interval (1, 5));
interval::Ivl 1 5
> abs (interval (-1, 5));
interval::Ivl 0 5
> abs (interval (-5, -1));
interval::Ivl 1 5
```

sgn v
: returns the interval representing the range of sgn(x) as x varies over v.

v
: returns the length of an interval.

Example 41 Absolute interval:

```
> #d;
4
```

(The symbol d was defined in Example 36.)

18.11.2 Least Complex Approximation within Epsilon

rational::**rational_approx_epsilon** ϵ r
: Find the least complex (see Complexity Extrema) rational approximation to r (usually a double) that is ϵ-close. That is find the q with the smallest possible denominator such that such that $|q - r| \leq \epsilon$. ($\epsilon > 0$.)

Example 42 Rational approximation to $\pi \approx 3.142 \approx 22/7$:

```
> rational_approx_epsilon .01 pi;
22L%7L
> abs (ans - pi);
0.00126448926734968
```

Example 43 The golden ratio $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$:

```
> let phi = (1 + sqrt 5) / 2;
> rational_approx_epsilon .001 phi;
55L%34L
> abs (ans - phi);
0.000386929926365465
```

`rational::`**`rational_approxs_epsilon`** ϵ r
: Produce a list of ever better rational approximations to r (usually a double) that is eventually ϵ-close. ($\epsilon > 0$.)

Example 44 Rational approximations to π:

```
> rational_approxs_epsilon .0001 pi;
[3L%1L,25L%8L,47L%15L,69L%22L,91L%29L,113L%36L,135L%43L,157L%50L,179L%57L,
201L%64L,223L%71L,245L%78L,267L%85L,289L%92L,311L%99L,333L%106L]
```

Example 45 Rational approximations to the golden ratio ϕ; these approximations are always reverse consecutive Fibonacci numbers (from f1: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...):

```
> rational_approxs_epsilon .0001 phi;
[1L%1L,3L%2L,8L%5L,21L%13L,55L%34L,144L%89L]
```

(The symbol phi was defined in Example 43.)

`rational::`**`rational_interval_epsilon`** ϵ r
: Find the least complex (see Complexity Extrema) rational interval containing r (usually a double) that is ϵ-small. That is find the least complex (see Complexity Extrema) q1 $\leq$ q2 such that r $\in$ [q1, q2] and q2 $-$ q1 $\leq \epsilon$. ($\epsilon > 0$.)

Example 46 Rational interval surrounding π:

```
> let i_Pi = rational_interval_epsilon .01 pi;
> i_Pi;
interval::Ivl (47L%15L) (22L%7L)
> double (lower i_Pi); pi; double (upper i_Pi);
3.13333333333333
3.14159265358979
3.14285714285714
```

(The functions lower and upper are described in Interval Constructors and 'Deconstructors'.)

Example 47 Rational interval surrounding the golden ratio ϕ:

```
> rational_interval_epsilon .001 phi;
interval::Ivl (55L%34L) (89L%55L)
> #ans;
1L%1870L
```

(The symbol phi was defined in Example 43. The function # is described in Interval Maths.)

18.11.3 Best Approximation with Bounded Denominator

`rational::`**`rational_approx_max_den`** maxDen r
: Find the closest rational approximation to r (usually a double) that has a denominator no greater than maxDen. (maxDen > 0).

Example 48 Rational approximation to π:

```
> rational_approx_max_den 10 pi;
22L%7L
```

Example 49 Rational approximation to the golden ratio ϕ:

```
> rational_approx_max_den 1000 phi;
1597L%987L
```

(The symbol phi was defined in Example 43.)

`rational::`**`rational_approxs_max_den`** maxDen r
: Produce a list of ever better rational approximations to r (usually a double) while the denominator is bounded by maxDen (maxDen > 0).

Example 50 Rational approximations to π:

```
> rational_approxs_max_den 100 pi;
[3L%1L,25L%8L,47L%15L,69L%22L,91L%29L,113L%36L,135L%43L,157L%50L,179L%57L,
201L%64L,223L%71L,245L%78L,267L%85L,289L%92L,311L%99L]
```

Example 51 Rational approximations to the golden ratio ϕ:

```
> rational_approxs_max_den 100 phi;
[1L%1L,3L%2L,8L%5L,21L%13L,55L%34L,144L%89L]
```

(The symbol phi was defined in Example 43.)

`rational::`**`rational_interval_max_den`** maxDen r
: Find the smallest rational interval containing r (usually a double) that has endpoints with denominators no greater than maxDen (maxDen > 0).

Example 52 Rational interval surrounding π:

```
> let i_Pi = rational_interval_max_den 100 pi ; i_Pi;
interval::Ivl (311L%99L) (22L%7L)
> double (lower i_Pi); pi; double (upper i_Pi);
3.14141414141414
3.14159265358979
3.14285714285714
```

Example 53 Rational interval surrounding the golden ratio ϕ:

```
> rational_interval_max_den 1000 phi;
interval::Ivl (987L%610L) (1597L%987L)
```

(The symbol phi was defined in Example 43.)

To approximate for a particular (rather than bounded) denominator, use round to multiple instead (see Rounding to Multiples).

18.12 Decomposition

There is more than one way to 'decompose' a rational number into its 'components'. It might be split into an integer and a fraction part — see Integer and Fraction Parts; or sign and absolute value — see Absolute Value and Sign; or numerator and denominator — see 'Deconstructors'.

18.13 Continued Fractions

18.13.1 Introduction

In "pure-rational", a continued fraction $a_0 + (1 / (a_1 + (1 / (a_2 + \cdots + 1 / a_n))))$ where $\forall i > 0 \bullet a_i \neq 0$, is represented by $[a_0, a_1, a_2, \ldots, a_n]$.

A 'simple' continued fraction is one in which $\forall i \bullet a_i \in \mathbf{Z}$ and $\forall i > 0 \bullet a_i > 0$.

Simple continued fractions for rationals are not quite unique since $[a_0, a_1, \ldots, a_n, 1] = [a_0, a_1, \ldots, a_{n+1}]$. We will refer to these as the 'non-standard' and 'standard' forms, respectively. The following functions return the standard form.

18.13.2 Generating Continued Fractions

Exact

`rational::`**`continued_fraction`** q
: Find 'the' (exact) continued fraction of a rational (including, trivially, integer) value q.

Example 54 The rational 1234/1001:

```
> continued_fraction (1234%1001);
[1L,4L,3L,2L,1L,1L,1L,8L]
> evaluate_continued_fraction ans;
1234L%1001L
```

Inexact

`rational::`**`continued_fraction_max_terms`** n r
: Find up to n initial terms of continued fraction of the value r with the 'remainder', if any, in the final element. (If continued_fraction_max_terms n r returns a list of length n or less, then the result is exact.)

Example 55 First 5 terms of the continued fraction for the golden ratio ϕ:

```
> continued_fraction_max_terms 5 phi;
  [1.0,1.0,1.0,1.0,1.0,1.61803398874989]
```

```
> evaluate_continued_fraction ans;
1.61803398874989
```

(The symbol phi was defined in Example 43.)

`rational::`**`continued_fraction_epsilon`** ϵ r
: Find enough of the initial terms of a continued fraction to within ϵ of the value r with the 'remainder', if any, in the final element.

Example 56 First few terms of the value $\sqrt{2}$:

```
> continued_fraction_epsilon .001 (sqrt 2);
[1.0,2.0,2.0,2.0,2.0,2.41421356237241]
> map double (convergents ans);
[1.0,1.5,1.4,1.41666666666667,1.41379310344828,1.41421356237309]
```

18.13.3 Evaluating Continued Fractions

`rational::`**`evaluate_continued_fraction`** aa
: Fold a continued fraction aa into the value it represents. This function is not limited to simple continued fractions. (Exact simple continued fractions are folded into a rational.)

Example 57 The continued fraction [1, 2, 3, 4] and the non-standard form [4, 3, 2, 1]:

```
> evaluate_continued_fraction [1,2,3,4];
43L%30L
> continued_fraction ans;
[1L,2L,3L,4L]
> evaluate_continued_fraction [4,3,2,1];
43L%10L
> continued_fraction ans;
[4L,3L,3L]
```

Convergents

`rational::`**`convergents`** aa
: Calculate the convergents of the continued fraction aa. This function is not limited to simple continued fractions.

Example 58 Convergents of a continued fraction approximation of the value $\sqrt{2}$:

```
> continued_fraction_max_terms 5 (sqrt 2);
[1.0,2.0,2.0,2.0,2.0,2.41421356237241]
> convergents ans;
[1.0,1.5,1.4,1.41666666666667,1.41379310344828,1.41421356237309]
```

18.14 Rational Complex Numbers

Pure together with `rational.pure` provide various types of number, including integers (**Z**), doubles (**R**, roughly), complex numbers (**C**) and Gaussian integers (**Z**[i]), rationals (**Q**) and rational complex numbers (**Q**[i]).

Functions concerning 'complex numbers' are named with 'comp', whereas functions concerning 'complexity' (see Complexity) are named with 'cplx'.

18.14.1 Rational Complex Constructors and 'Deconstructors'

Complex numbers can have rational parts.

Example 59 Forming a rational complex:

```
> 1 +: 1 * (1%2);
1+:1L%2L
> ans * ans;
3L%4L+:1L%1L
```

And rational numbers can be given complex parts.

Example 60 Complex rationals and complicated rationals:

```
> (1 +: 2) % (3 +: 4);
11L%25L+:2L%25L
> ans * (3 +: 4);
1L%1L+:2L%1L
> ((4%1) * (0 +: 1)) % 2;
0L%1L+:2L%1L
> ((4%1) * (0 +: 1)) % (1%2);
0L%1L+:8L%1L
> ((4%1) * (0 +: 1)) % (1 + (1%2) * (0 +: 1));
8L%5L+:16L%5L
> ans * (1+(1%2) * (0 +: 1));
0L%1L+:4L%1L
> ((4%1) * (0 +: 1)) / (1 + (1%2) * (0 +: 1));
1.6+:3.2
```

The various parts of a complex rational may be deconstructed using combinations of num and den and the standard functions re and im.

Thus, taking real and imaginary parts first, a rational complex number may be considered to be $(x_n / x_d) + (y_n / y_d) * i$ with $x_n, x_d, y_n, y_d \in \mathbf{Z}$.

A rational complex number may also be decomposed into its 'numerator' and 'denominator', where these are both integral complex numbers, or 'Gaussian integers', and the denominatoris a minimal choice in some sense.

One way to do this is so that the denominator is the minimum positive integer. The denominator is a complex number with zero imaginary part.

Thus, taking numerator and denominator parts first, a rational complex number may be considered to be (n_x + n_y * i) / (d + 0 * i) with n_x, n_y, d ∈ **Z**.

Another way to do this is so that the denominator is a Gaussian integer with minimal absolute value. Thus, taking numerator and denominator parts first, a rational complex number may be considered to be (n_x + n_y * i) / (d_x + d_y * i) with n_x, n_y, d_x, d_y ∈ **Z**.

The d_x, d_y are not unique, but can be chosen such that $d_x > 0$ and either $|d_y| < d_x$ or $d_y = d_x > 0$.

`rational::`**`num_den_nat`** c
: given a complex rational or integer c, returns a pair (n, d) containing an integral complex (Gaussian integral) numerator n, and the smallest natural (i.e. positive integral real) complex denominator d, i.e. a complex number where ℜ(d) ∈ **Z**, ℜ(d) > 0, ℑ(d) = 0; i.e. the numerator and denominator of one 'normalised' form of c.

  This is an inverse (up to equivalence) of rational as defined on integral complex pairs (see Constructors).

`rational::`**`num_den_gauss`** c
: given a complex rational or integer c, returns a pair (n, d) containing an integral complex (Gaussian integral) numerator n, and an absolutely smallest integral complex denominator d chosen s.t. ℜ(d),=(d) ∈ **Z**, ℜ(d) > 0, and either |ℜ(d)| < ℑ(d) or ℜ(d) = ℑ(d) > 0; i.e. the numerator and denominator of another 'normalised' form of c.

  This is an inverse (up to equivalence) of rational as defined on integral complex pairs (see Constructors).

`rational::`**`num_den`** c
: synonymous with num_den_gauss.

  This is an inverse (up to equivalence) of rational as defined on integer pairs (see Constructors).

`num` c
: given a complex rational or integer c, returns just the numerator of the normalised form of c given by num_den c.

`den` c
: given a complex rational or integer c, returns just the denominator of the normalised form of c given by num_den c.

Example 61 Rational complex number deconstruction:

```
> let cq = (1+2*i)%(3+3*i); cq;
1L%2L+:1L%6L
> (re cq, im cq);
1L%2L,1L%6L
> (num . re) cq;
1L
> (den . re) cq;
2L
> (num . im) cq;
```

```
1L
> (den . im) cq;
6L
> let (n_nat,d_nat) = num_den_nat cq;
> (n_nat, d_nat);
3+:1,6+:0
> n_nat % d_nat;
1L%2L+:1L%6L
> abs d_nat;
6.0
> let (n, d) = num_den_gauss cq; (n, d);
1L+:2L,3L+:3L
> let (n,d) = num_den cq; (n, d);
1L+:2L,3L+:3L
> n % d;
1L%2L+:1L%6L
> abs d;
4.24264068711928
> (re . num) cq;
1L
> (im . num) cq;
2L
> (re . den) cq; //always > 0
3L
> (im . den) cq; //always <= (re.den)
3L
```

18.14.2 Rational Complex Type and Value Tests

Beware that `intcompvalp` and `ratcompvapl` may return 1 even if the value is of complex type with double parts. However, these functions may be combined with `exactp`.

`complexp` x
: standard function; returns whether x is of complex type.

`compvalp` x
: standard function; returns whether x has a complex value ($\in \mathbf{C} = \mathbf{R}[i]$).

`rational::`**`ratcompvalp`** x
: returns whether x has a rational complex value ($\in \mathbf{Q}[i]$).

`rational::`**`intcompvalp`** x
: returns whether x has an integral complex value ($\in \mathbf{Z}[i]$), i.e. a Gaussian integer value.

Example 62 Rational complex number value tests:

```
> let l = [9, 9%1, 9%2, 4.5, sqrt 2, 1+:1, 1%2+:1, 0.5+:1, inf, nan];
> map exactp l;
[1,1,1,0,0,1,1,0,0,0]
> map inexactp l;
[0,0,0,1,1,0,0,1,1,1]
```

```
> map complexp l;
[0,0,0,0,0,1,1,1,0,0]
> map compvalp l;
[1,1,1,1,1,1,1,1,1,1]
> map (\x -> (exactp x and compvalp x)) l; // "has exact complex value"
[1,1,1,0,0,1,1,0,0,0]
> map ratcompvalp l;
[1,1,1,1,1,1,1,1,0,0]
> map (\x -> (exactp x and ratcompvalp x)) l;
[1,1,1,0,0,1,1,0,0,0]
> map intcompvalp l;
[1,1,0,0,0,1,0,0,0,0]
> map (\x -> (exactp x and intcompvalp x)) l;
[1,1,0,0,0,1,0,0,0,0]
> map ratvalp l;
[1,1,1,1,1,0,0,0,0,0]
> map (\x -> (exactp x and ratvalp x)) l;
[1,1,1,0,0,0,0,0,0,0]
> map intvalp l; // for comparison
[1,1,0,0,0,0,0,0,0,0]
> map (\x -> (exactp x and intvalp x)) l;
[1,1,0,0,0,0,0,0,0,0]

See 'Type and Value Tests'_ for some details of rational type and value tests.
```

18.14.3 Rational Complex Arithmetic Operators and Relations

The standard arithmetic operators (+), (−), (*), (/), (%), (), (==) and (~=) are overloaded to have at least one complex and/or rational operand, but (<), (<=), (>), (>=) are not, as complex numbers are unordered.

Example 63 Rational complex arithmetic:

```
> let w = 1%2 +: 3%4;
> let z = 5%6 +: 7%8;
> w + z;
4L%3L+:13L%8L
> w % z;
618L%841L+:108L%841L
> w / z;
0.734839476813318+:0.128418549346017
> w ^ 2;
-0.3125+:0.75
> w == z;
0
> w == w;
1
```

18.14.4 Rational Complex Maths

The standard functions `re` and `im` work with rational complex numbers (see Rational Complex Constructors and 'Deconstructors').

The standard functions `polar`, `abs` and `arg` work with rational complex numbers, but the results are inexact.

Example 64 Rational complex maths:

```
> polar (1%2+:1%2);
0.707106781186548<:0.785398163397448
> abs (4%2+:3%2);
2.5
> arg (-1%1);
3.14159265358979
```

There are some additional useful functions for calculating with rational complex numbers and more general mathematical values.

`rational::`**`norm_gauss`** c
: returns the Gaussian norm ||c|| of any complex (or real) number c; this is the square of the absolute value, and is returned as an (exact) integer.

`rational::`**`div_mod_gauss`** n d
: performs Gaussian integer division, returning (q, r) where q is a (not always unique) quotient, and r is a (not always unique) remainder. q and r are such that n = q * d + r and ||r|| < ||d|| (equivalently, |r| < |d|).

`rational::`**`n_div_gauss`** d
: returns just a quotient from Gaussian integer division as produced by div_mod_gauss n d.

`rational::`**`n_mod_gauss`** d
: returns just a remainder from Gaussian integer division as produced by div_mod_gauss n d.

`rational::`**`gcd_gauss`** c1 c2
: returns a GCD G of the Gaussian integers c1,c2. This is chosen so that s.t. $\Re(G) > 0$, and either $|\Im(G)| < \Re(G)$ or $\Im(G) = \Re(G) > 0$;

`rational::`**`euclid_gcd`** zerofun modfun x y
: returns a (non-unique) GCD calculated by performing the Euclidean algorithm on the values x and y (of any type) where zerofun is a predicate for equality to 0, and modfun is a binary modulus (remainder) function.

`rational::`**`euclid_alg`** zerofun divfun x y
: returns (g, a, b) where the g is a (non-unique) GCD and a, b are (arbitrary, non-unique) values such that a * x + b * y = g calculated by performing the generalised Euclidean algorithm on the values x and y (of any type) where zerofun is a predicate for equality to 0, and div is a binary quotient function.

Example 65 More rational complex and other maths:

```
> norm_gauss (1 +: 3);
10
> abs (1 +: 3);
3.16227766016838
> norm_gauss (-5);
25
> let (q, r) = div_mod_gauss 100 (12 +: 5);
> (q, r);
7L+:-3L,1L+:1L
> q * (12 +: 5) + r;
100L+:0L
> 100 div_gauss (12 +: 5);
7L+:-3L
> 100 mod_gauss (12 +: 5);
1L+:1L
> div_mod_gauss 23 5;
5L+:0L,-2L+:0L
> gcd_gauss (1 +: 2) (3 +: 4);
1L+:0L
> gcd_gauss 25 15;
5L+:0L
> euclid_gcd (==0) (mod_gauss) (1+: 2) (3 +: 4);
1L+:0L
> euclid_gcd (==0) (mod) 25 15;
5
> let (g, a, b) = euclid_alg (==0) (div_gauss) (1 +: 2) (3 +: 4); g;
1L+:0L
> (a, b);
-2L+:0L,1L+:0L
> a * (1 +: 2) + b * (3 +: 4);
1L+:0L
> let (g, a, b) = euclid_alg (==0) (div) 25 15; g;
5
> (a, b);
-1,2
> a * 25 + b * 15;
5
```

18.14.5 Rational Complex Type Simplification

`rational::`**`comp_simplify`** c
: returns q with complex numbers simplified to reals, if possible.

`rational::`**`ratcomp_simplify`** c
: returns q with rationals simplified to integers, and complex numbers simplified to reals, if possible.

Example 66 Rational complex number type simplification:

```
> let l = [9+:1, 9%1+:1, 9%2+:1, 4.5+:1, 9%1+:0, 9%2+:0, 4.5+:0.0];
> l;
```

```
[9+:1,9L%1L+:1,9L%2L+:1,4.5+:1,9L%1L+:0,9L%2L+:0,4.5+:0.0]
> map comp_simplify l;
[9+:1,9L%1L+:1,9L%2L+:1,4.5+:1,9L%1L,9L%2L,4.5+:0.0]
> map ratcomp_simplify l;
[9+:1,9+:1,9L%2L+:1,4.5+:1,9,9L%2L,4.5+:0.0]

See 'Rational Type Simplification'_ for some details of rational type
simplification.
```

18.15 String Formatting and Evaluation

18.15.1 The Naming of the String Conversion Functions

There are several families of functions for converting between strings and rationals.

The functions that convert from rationals to strings have names based on that of the standard function `str`. The `str_*` functions convert to a formatted string, and depend on a 'format structure' parameter (see Internationalisation and Format Structures). The `strs_*` functions convert to a tuple of string fragments.

The functions that convert from strings to rationals have names based on that of the standard function `eval` (`val` in Q). The `val_*` functions convert from a formatted string, and depend on a format structure parameter. The `sval_*` functions convert from a tuple of string fragments.

There are also `join_*` and `split_*` functions to join string fragments into formatted strings, and to split formatted strings into string fragments, respectively; these depend on a format structure parameter. These functions are not always invertible, because some of the functions reduce an error term to just a sign, e.g. `str_real_approx_dp` may round a value. Thus sometimes the `join_*` and `split_*` pairs, and the `str_*` and `val_*` pairs are not quite mutual inverses.

18.15.2 Internationalisation and Format Structures

Many of the string formatting functions in the following sections are parameterised by a 'format structure'. Throughout this document, the formal parameter for the format structure will be `fmt`. This is simply a record mapping some string 'codes' to functions as follows. The functions are mostly from strings to a string, or from a string to a tuple of strings.

`"sm"` a function mapping a sign and an unsigned mantissa (or integer) strings to a signed mantissa (or integer) string.

`"se"` a function mapping a sign and an unsigned exponent string to a signed exponent string.

`"-s"` a function mapping a signed number string to a pair containing a sign and the unsigned number string.

"gi" a function mapping an integer representing the group size and an integer string to a grouped integer string.

"gf" a function mapping an integer representing the group size and a fraction-part string to a grouped fraction-part string.

"-g" a function mapping a grouped number string to an ungrouped number string.

"zi" a function mapping an integer number string to a number string. The input string representing zero integer part is "", which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

"zf" a function mapping a fraction-part number string to a number string. The input string representing zero fraction part is "", which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

"ir" a function mapping initial and recurring parts of a fraction part to the desired format.

"-ir" a function mapping a formatted fraction part to the component initial and recurring parts.

"if" a function mapping an integer string and fraction part string to the radix-point formatted string.

"-if" a function mapping a radix-point formatted string to the component integer fraction part strings

"me" a function mapping a mantissa string and exponent string to the formatted exponential string.

"-me" a function mapping a formatted exponential string to the component mantissa and exponent strings.

"e" a function mapping an 'error' number (not string) and a number string to a formatted number string indicating the sign of the error.

"-e" a function mapping a formatted number string indicating the sign of the error to the component 'error' string (not number) and number strings.

Depending upon the format structure, some parameters of some of the functions taking a format structure may have no effect. For example, an `intGroup` parameter specifying the size of the integer digit groups will have no effect if the integer group separator is the empty string.

`rational::`**`create_format`** options
: is a function that provides an easy way to prepare a 'format structure' from the simpler 'options structure'. The options structure is another record, but from more descriptive strings to a string or tuple of strings.

For example, `format_uk` is generated from `options_uk` as follows:

```
public options_uk;
const options_uk =
  {
    "sign" => ("-","",""),            // alternative: ("-"," ","+")
```

```
    "exponent sign" => ("-","",""),    // alternative: ("-","","+")
    "group separator" => ",",          // might be " " or "." or "'" elsewhere
    "zero" => "0",
    "radix point" => ".",              // might be "," elsewhere
    "fraction group separator" => ",",
    "fraction zero" => "0",            // alternative: ""
    "recur brackets" => ("[","...]"),
    "exponent" => "*10^",              // (poor) alternative: "e"
    "error sign" => ("-","","+"),
    "error brackets" => ("(",")")
  };

public format_uk;
const format_uk = create_format options_uk;
```

The exponent string need not depend on the radix, as the numerals for the number radix in that radix are always "10".

Beware of using "e" or "E" as an exponent string as these have the potential of being treated as digits in, e.g., hexadecimal.

Format structures do not have to be generated via create format; they may also be constructed directly.

18.15.3 Digit Grouping

Some functions take `group` parameters. A value of 0 means "don't group".

18.15.4 Radices

The functions that produce a decimal expansion take a Radix argument. The fraction parts are expanded in that radix (or 'base'), in addition to the integer parts. The parameter Radix is not restricted to the usual {2, 8, 10, 16}, but may be any integer from 2 to 36; the numerals ('digits') are chosen from ["0", ... , "9", "A", ... , "Z"]. The letter-digits are always upper case.

The functions do not attach a prefix (such as "0x" for hexadecimal) to the resulting string.

18.15.5 Error Terms

Some functions return a value including an 'error' term (in a tuple) or sign (at the end of a string). Such an error is represents what the next digit would be as a fraction of the radix.

Example 67 Error term in the tuple of string 'fragments':

```
> strs_real_approx_sf 10 floor 3 (234567%100000);
"+","2","34",567L%1000L
> strs_real_approx_sf 10 ceil 3 (234567%100000);
"+","2","35",(-433L)%1000L
```

(See the function strs_real_approx_sf.)

In strings, only the sign of the error term is given. A "+" should be read as "and a bit more"; "-" as "but a bit less".

Example 68 Error sign in the string:

```
> str_real_approx_sf format_uk 10 0 0 floor 3 (234567%100000);
"2.34(+)"
> str_real_approx_sf format_uk 10 0 0 ceil 3 (234567%100000);
"2.35(-)"
```

(See the function str_real_approx_sf.)

18.16 Q <-> Fraction String ("i + n/d")

18.16.1 Formatting to Fraction Strings

rational::**str_vulgar** q
: returns a String representing the rational (or integer) q in the form

  - "[−]n/d"

rational::**str_vulgar_or_int** q
: returns a String representing the rational (or integer) q in one of the forms

  - "[−]n/d"
  - "[−]i"

rational::**str_mixed** q
: returns a String representing the rational (or integer) q in one of the forms

  - "i + n/d"
  - "−(i + n/d)"
  - "[−]n/d"
  - "[−]i"

Example 69 The fraction string representations:

```
> let l = iterwhile (<= 3%2) (+(1%2)) (-3%2);
> l;
[(-3L)%2L,(-1L)%1L,(-1L)%2L,0L%1L,1L%2L,1L%1L,3L%2L]
> map str_vulgar l;
["-3L/2L","-1L/1L","-1L/2L","0L/1L","1L/2L","1L/1L","3L/2L"]
> map str_vulgar_or_int l;
["-3L/2L","-1L","-1L/2L","0L","1L/2L","1L","3L/2L"]
> map str_mixed l;
["-(1L+1L/2L)","-1L","-1L/2L","0L","1L/2L","1L","1L+1L/2L"]
```

These might be compared to the behaviour of the standard function str.

str x
: returns a string representing the value x.

Example 70 The standard function str:

```
> map str l;
["(-3L)%2L","(-1L)%1L","(-1L)%2L","0L%1L","1L%2L","1L%1L","3L%2L"]
```

18.16.2 Evaluation of Fraction Strings

rational::**val_vulgar** strg
: returns a rational q represented by the string strg in the form

  - "[−]n/d"

Such strings can also be evaluated by the val_mixed function.

rational::**val_mixed** strg
: returns a rational q represented by the string strg

  - "i + n/d"
  - "−(i + n/d)"
  - "[−]n/d" — thus val_mixed strictly extends val_vulgar
  - "[−]i"

Example 71 Evaluating fraction strings:

```
> val_vulgar "-22/7";
(-22L)%7L
> val_mixed "1L+5L/6L";
11L%6L
```

These might be compared to the behaviour of the standard function eval.

eval s
: evaluates the string s.

Example 72 The standard function eval:

```
> eval "1+5%6";
11L%6L
> eval "1L+5L/6L";
1.83333333333333
```

18.17 Q <-> Recurring Numeral Expansion String ("I.FR")

See Internationalisation and Format Structures for information about the formatting structure to be supplied in the fmt parameter.

18.17.1 Formatting to Recurring Expansion Strings

`rational::`**`str_real_recur`** fmt radix intGroup q
: returns a string (exactly) representing the rational (or integer) q as base-Radix expansion of one the forms

  - "[−]int.frac"
  - "[−]int.init frac part[smallest recurring frac part ...]"

Note that there is no fracGroup parameter.

Beware that the string returned by this function can be very long. The length of the recurring part of such a decimal expansion may be up to one less than the simplest denominator of q.

Example 73 The recurring radix expansion-type string representations:

```
> str_real_recur format_uk 10 3 (4000001%4); // grouped with commas
"1,000,000.25"
> str_real_recur format_uk 10 0 (4000001%4); // no grouping
"1000000.25"
> str_real_recur format_uk 10 3 (1000000%3);
"333,333.[3...]"
> str_real_recur format_uk 10 3 (1000000%7);
"142,857.[142857...]"
> str_real_recur format_uk 10 3 (-1%700);
"-0.00[142857...]"
> str_real_recur format_uk 10 3 (127%128);
"0.9921875"
> str_real_recur format_uk 2 4 (-127%128);
"-0.1111111"
> str_real_recur format_uk 16 4 (127%128);
"0.FE"
> str_real_recur format_uk 10 0 (70057%350); // 1%7 + 10001%50;
"200.16[285714...]"
```

The function allows expansion to different radices (bases).

Example 74 The recurring radix expansion in decimal and hexadecimal:

```
> str_real_recur format_uk 10 0 (1%100);
"0.01"
> str_real_recur format_uk 16 0 (1%100);
"0.0[28F5C...]"
```

Example 75 The recurring radix expansion in duodecimal:

```
> str_real_recur format_uk 12 0 (1%100);
"0.0[15343A0B62A68781B059...]"
```

Note that this bracket notation is not standard in the literature. Usually the recurring numerals are indicated by a single dot over the initial and final numerals of the recurring part, or an overline over the recurring part. For example $1/70 = 0.0\dot{1}4285\dot{7} = 0.0142857$ and $1/3 = 0.\dot{3} = 0.3$.

`rational::`**`strs_real_recur`** radix q
: returns a quadruple of the four strings:

  - the sign,
  - integer part (which is empty for 0),
  - initial fraction part
  - and recurring fraction part (either and both of which may be empty).

Example 76 The recurring radix expansion in decimal — the fragments:

```
> strs_real_recur 10 (100%7);
"+","14","","285714"
> strs_real_recur 10 (-1%700);
"-","","00","142857"
> strs_real_recur 10 (70057%350);
"+","200","16","285714"
```

This function may be used to also, e.g. format the integer part with comma-separated groupings.

`rational::`**`join_str_real_recur`** fmt intGroup sign i fracInit fracRecur
: formats the parts in the quadruple returned by `strs_real_recur` to the sort of string as returned by `str_real_recur`.

18.17.2 Evaluation of Recurring Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between rationals and differently formatted strings if a suitable alternative format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

`rational::`**`val_real_recur`** fmt radix strg
: returns the rational q represented by the base-radix expansion string strg of one the forms

  - "[−]int.frac"
  - "[−]int.init frac part[recurring frac part ...]"

Example 77 Conversion from the recurring radix expansion-type string representations:

```
> val_real_recur format_uk 10 "-12.345";
(-2469L)%200L
> val_real_recur format_uk 10 "0.3";
3L%10L
> val_real_recur format_uk 10 "0.[3...]";
1L%3L
> val_real_recur format_uk 10 ".333[33...]";
1L%3L
```

```
> val_real_recur format_uk 10 ".[9...]";
1L%1L
```

`rational::`**`sval_real_recur`** radix sign iStr fracStr recurPartStr
: returns the rational q represented by the parts

  - sign
  - integer part
  - initial fraction part
  - recurring fraction part

`rational::`**`split_str_real_recur`** Fmt strg
: returns a tuple containing the parts

  - sign
  - integer part
  - initial fraction part
  - recurring fraction part of one the forms - "[−]int.frac" - "[−]int.init frac part[recurring frac part ...]"

18.18 Q <-> Numeral Expansion String ("I.F × 10E")

See Internationalisation and Format Structures for information about the formatting structure to be supplied in the `fmt` parameter.

The exponent string "*10^" need not depend on the radix, as the numerals for the number radix in that radix are always "10".

18.18.1 Formatting to Expansion Strings

Functions for Fixed Decimal Places

`rational::`**`str_real_approx_dp`** fmt radix intGroup fracGroup roundfun dp q
: returns a string representing a numeral expansion approximation of q to dp decimal places, using rounding mode `roundfun` (see Rounding to Integer) `roundfun` is usually `round` or `round_unbiased`. (dp may be positive, zero or negative; non-positive dps may look misleading — use e.g. scientific notation instead.)

Example 78 Decimal places:

```
> str_real_approx_dp format_uk 10 3 3 round 2 (22%7);
"3.14(+)"
> str_real_approx_dp format_uk 10 3 3 ceil 2 (22%7);
"3.15(-)"
```

`rational::`**`strs_real_approx_dp`** radix roundfun do q
: returns a tuple of strings

  - sign
  - integer part
  - fraction part

  representing an expansion to a number of decimal places, together with

  - the rounding "error": a fraction representing the next numerals.

Example 79 Decimal places — the fragments:

```
> strs_real_approx_dp 10 round 2 (22%7);
"+","3","14",2L%7L
> strs_real_approx_dp 10 ceil 2 (22%7);
"+","3","15",(-5L)%7L
```

`rational::`**`join_str_real_approx`** fmt intGroup fracGroup sign i frac err
: formats the parts in the quadruple returned by `strs_real_approx_dp` or `strs_real_approx_sf` to the sort of string as returned by `str_real_approx_dp` or `str_real_approx_sf`.

Functions for Significant Figures

`rational::`**`str_real_approx_sf`** fmt radix intGroup fracGroup roundfun sf q
: returns a string representing a numeral expansion approximation of q to sf significant figures, using rounding mode `roundfun` (see Rounding to Integer).

`roundfun` is usually `round` or `round_unbiased`. (sf must be positive.)

Example 80 Significant figures:

```
> str_real_approx_sf format_uk 10 3 3 floor 2 (22%7);
"3.1(+)"
> str_real_approx_sf format_uk 10 3 3 floor 2 ((-22)%7);
"-3.2(+)"
```

`rational::`**`strs_real_approx_sf`** radix roundfun sf q
: returns a tuple of strings

  - sign,
  - integer part,
  - fraction part, representing an expansion to a number of significant figures, together with
  - the rounding "error": a fraction representing the next numerals

`rational::`**`join_str_real_approx`**
: see `join_str_real_approx`.

Functions for Scientific Notation and Engineering Notation

`rational::`**`str_real_approx_sci`** fmt radix intGroup fracGroup roundfun sf q
: returns a string expansion with a number of significant figures in scientific notation, using rounding mode `roundfun` (see Rounding to Integer).

  (sf must be positive; expStep is usually 3, radix is usually 10, `roundfun` is usually `round` or `round_unbiased`; `str_real_approx_sci` is equivalent to `str_real_approx_eng` (below) with expStep = 1.)

`rational::`**`strs_real_approx_sci`** radix roundfun sf q
: returns a tuple of strings:

  - sign of mantissa,
  - integer part of mantissa,
  - fraction part of mantissa,
  - sign of exponent,
  - exponent magnitude

  representing an expansion to a number of significant figures in scientific notation together with

  - the rounding "error": a fraction representing the next numerals.

`rational::`**`str_real_approx_eng`** fmt expStep radix intGroup fracGroup round sf q
: returns a string expansion with a number of significant figures in engineering notation, using rounding mode roundfun.

  The ExpStep parameter specifies the granularity of the exponent; specifically, the exponent will always be divisible by expStep.

  (sf must be positive; expStep is usually 3 and must be positive, radix is usually 10, `roundfun` is usually `round` or `round_unbiased`.)

Example 81 Engineering notation:

```
> str_real_approx_eng format_uk 3 10 3 3 round 7 (rational 999950);
"999.950,0*10^3"
> str_real_approx_eng format_uk 3 10 3 3 round 4 999950;
"1.000*10^6(-)"
```

`rational::`**`strs_real_approx_eng`** expStep radix roundfun sf q
: returns a tuple of strings:

  - sign of mantissa,
  - integer part of mantissa,
  - fraction part of mantissa,
  - sign of exponent,

- exponent magnitude

representing an expansion to a number of significant figures in engineering notation together with

- the rounding "error": a fraction representing the next numerals.

Example 82 Engineering notation — the fragments:

```
> strs_real_approx_eng 3 10 round 7 (rational 999950);
"+","999","9500","+","3",0L%1L
> strs_real_approx_eng 3 10 round 4 999950;
"+","1","000","+","6",(-1L)%20L
```

`rational::`**`join_str_real_eng`** fmt intGroup fracGroup mantSign mantI mantF rac expSign expI err
formats the parts in the quadruple returned by `strs_real_approx_eng` or strs_real_approx_sci to the sort of string as returned by `str_real_approx_eng` or `str_real_approx_sci`.

18.18.2 Evaluation of Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between rationals and differently formatted strings if a suitable alternative format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

`rational::`**`val_real_eng`** fmt radix strg
returns the rational q represented by the base-radix expansion string strg of one the forms

- "[−]int.frac"
- "[−]int.frace[−]exponent"

Example 83 Conversion from the recurring radix expansion-type string representations:

```
> val_real_eng format_uk 10 "-12.345";
(-2469L)%200L
> val_real_eng format_uk 10 "-12.345*10^2";
(-2469L)%2L
```

`rational::`**`sval_real_eng`** radix signStr mantIStr mantF racStr expSignStr expStr
returns the rational q represented by the parts

- sign
- integer part of mantissa
- fraction part of mantissa
- sign of exponent
- exponent

`rational::`**`split_str_real_eng`** fmt strg
: returns a tuple containing the string parts

  - sign
  - integer part of mantissa
  - fraction part of mantissa
  - sign of exponent
  - exponent
  - the "error" sign

  of one the forms

  - "[−]int.frac"
  - "[−]int.frac ×10^[−]exponent"

These functions can deal with the fixed decimal places, the significant figures and the scientific notation in addition to the engineering notation.

18.19 Numeral String -> Q — Approximation

This section describes functions to approximate by a rational a real number represented by a string. See R -> Q — Approximation for approximation by a rational of a double.

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between rationals and differently formatted strings if a format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

`rational::`**`val_eng_approx_epsilon`** fmt radix epsilon strg
: Find the least complex rational approximation q to the number represented by the base-radix expansion string str in one of the forms

  - "[−]int.frac"
  - "[−]int.frac ×10^[−]exponent"

  that is ϵ-close. That is find a q such that $|q - \text{eval str}| \leq \epsilon$.

Example 84 Rational from a long string:

```
> let strg = "123.456,789,876,543,212,345,678,987,654,321*10^27";
> let x = val_real_eng format_uk 10 strg;
> x;
123456789876543212345678987654321L%1000L
> let q = val_eng_approx_epsilon format_uk 10 (1%100) strg;
> q;
19753086380246913975308638024691L%16L
> double (x - q);
```

```
0.0085
> str_real_approx_eng format_uk 3 10 3 3 round 30 q;
"123.456,789,876,543,212,345,678,987,654*10^27(+)"
> str_real_approx_eng format_uk 3 10 3 3 round 42 q;
"123.456,789,876,543,212,345,678,987,654,312,500,000,000*10^27"
> double q;
1.23456789876543e+029
```

`rational::`**`val_eng_interval_epsilon`** fmt radix epsilon strg
: Find the least complex rational interval containing the number represented by the base-radix expansion string strg in one of the forms

  - "[−]int.frac"
  - "[−]int.frac ×10^[−]exponent"

  that is "-small.

`rational::`**`val_eng_approx_max_den`** fmt radix maxDen strg
: Find the closest rational approximation to the number represented by the base-rRadix expansion string strg in one of the forms

  - "[−]int.frac"
  - "[−]int.frac ×10^[−]exponent"

  that has a denominator no greater than maxDen. (maxDen > 0)

`rational::`**`val_eng_interval_max_den`** fmt radix maxDen strg
: Find the smallest rational interval containing the number represented by the base-radix expansion string strg in one of the forms

  - "[−]int.frac"
  - "[−]int.frac ×10^[−]exponent"

  that has endpoints with denominators no greater than maxDen. (maxDen > 0)

Example 85 Other rationals from a long string:

```
> val_eng_approx_epsilon format_uk 10 (1%100) strg;
19753086380246913975308638024 69L%16L
> val_eng_interval_epsilon format_uk 10 (1%100) strg;
interval::Ivl (308641974691358030864197 4691358L%25L)
(345679011654320994567901165 4321L%28L)
> val_eng_approx_max_den format_uk 10 100 strg;
99999999800000001999999980000000L%81L
> val_eng_interval_max_den format_uk 10 100 strg;
interval::Ivl 99999999800000001999999980000000L%81L
345679011654320994567901165 4321L%28L
```

Chapter 19

Pure-CSV - Comma Separated Value Interface for the Pure Programming Language

Version 1.5, July 18, 2012

Eddie Rucker <erucker@bmc.edu>

The CSV library provides an interface for reading and writing comma separated value files. The module is very loosely based on Python's CSV module (http://docs.python.org/lib/module-csv.html).

19.1 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-csv-1.5.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make. The `make install` step is only necessary for system-wide installation.

The `make` utility tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. Please see the Makefile for details.

19.2 Usage

Data records are represented as vectors or lists of any Pure values. Values are converted as necessary and written as a group of strings, integers, or doubles separated by a delimiter. Three predefined dialects are provided; `DEFAULT` (record terminator= `\n`), `RFC4180` (record terminator= `\r\n`), and `Excel`. Procedures are provided to create other CSV dialects. See (http://www.ietf.org/rfc/rfc4180.txt) for more details about the RFC4180 standard.

19.2.1 Handling Errors

`error msg` is an error handling term. Operations resulting in parse errors, memory errors, or read/write errors produce a special `csv::error msg` term, where `msg` is a string describing the particular error. Your application should either check for these or have `csv::error` defined to directly handle errors in some way (e.g., provide a default value, or raise an exception).

19.2.2 Creating Dialects

`dialect record` creates a dialect from a record of dialect option pairs. The dialect object is freed automatically when exiting the pure script. The list of possible options and option values are presented below.

- `delimiter` - Character used to separate fields.
  - Value - any string.
  - Default - `","`.
- `escape` - Embedded escape character used to embed a delimiter, escape, or terminator into unquoted fields. If the escape character is not null, then the quote character is ignored.
  - Value - any string.
  - Default - `""`.
  - Reading - The escape character is dropped and the next char is inserted into the field.
  - Writing - The escape character is written into the output stream before the delimiter, escape, or return character.
- `quote` - Quotes are used to embed delimiters, quotes, or terminators into a field.
  - Value - any string.
  - Default - `"\""`.
  - Notes - Embedded quotes are doubled. The `escape` option must be the null string.
- `terminator` - Record termination string.
  - Value - any string.
  - Reading - Either a user specified string or if not specivied the file is sniffed for a `\r`, `\r\n`, or `\n`.
  - Writing - Either a user specified string, `\r\n` for Windows platforms, or `\n` for everything else.
- `quote_flag` - Sets the quoting style of strings and/or numbers.

  - Value - One of {`ALL`, `STRINGS`, `MINIMAL`}.
  - Default - `ALL`.
  - Reading -
    1. `ALL` - Every field is read as a string.
    2. `STRING`, `MINIMAL` - Fields within quotes and fields that cannot be converted to integers or doubles are read as strings.
  - Writing -
    1. `ALL` - Every field is written within quotes.
    2. `STRING` - Only fields of type `string` are quoted.
    3. `MINIMAL` - Only fields containing embedded quotes, terminators, or delimiters are written within quotes.

- `space_around_quoted_field` - Determines how white space between quotes and delimiters should be treated.
  - Value - One of {`NO`, `LEFT`, `RIGHT`, `BOTH`}.
  - Default - `NO`.
  - Reading -
    1. `NO` - Follows RFC4180 rules.
    2. `LEFT` - Allows space before a quoted field.
    3. `RIGHT` - Allows space between a quoted field and a delimiter.
    4. `BOTH` - Allows space before and after a quoted field.
  - Writing - fields are never written with space before a quoted field or between a quoted field and a delimiter.
  - Notes this option does not affect space within quotes or fields written using the `escape` string option.

- `trim_space` - trim white space before or after field contents.
  - Value - One of {`NO`, `LEFT`, `RIGHT`, `BOTH`}.
  - Default - `NO`.
  - Reading -
    1. `NO` - Reading follows RFC4180 rules.
    2. `LEFT`, `RIGHT`, or `BOTH` - The field is trimmed accordingly. Use *caution* because trimming may allow automatic conversion of numbers if the `quote_flag` is set to `MINIMAL`.
  - Writing -
    1. `NO` - Reading follows RFC4180 rules

2. `LEFT`, `RIGHT`, or `BOTH` - Trimming space is probably a bad idea since leading or trailing space may be significant for other applications.

The following example illustrates the construction of a dialect for reading tab delimited files without quoted strings.

Example

```
> using csv;
> using namespace csv;
> let d = dialect {delimiter=>"\t", quote_flag=>STRING};
>
```

19.2.3 Opening CSV Files

`open name::string` opens a CSV file for reading using the default dialect. If the file does not exist, the `error msg` rule is invoked.

`open (name::string, rw_flag::string)` opens a CSV file for reading, writing, or appending using the default dialect. Valid `rw_flag` values are `"r"` for reading, `"w"` for writing, and `"a"` for appending. If the file does not exist when opened for reading, the `error msg` rule is invoked. When a file is opened for writing and the file exists, the old file is overwritten. If the file does not exist, a new empty file is created. When a file is opened for appending and the file exists, new records are appended to the end of the file, otherwise a new empty file is created.

`open (name::string, rw_flag::string, d::matrix)` exactly as above except reading/writing is done according to a user defined dialect `d`.

`open (name::string, rw_flag::string, d::matrix, opts@(_:_))` exactly as above except allows for list output or header options when reading.

1. If `opts` contains `LIST`, the output of `getr`, `fgetr`, and `fgetr_lazy` is a list instead of a vector.

2. If `opts` contains `HEADER`, the first line of the file is automatically read and parsed as a record where entries are `key=>position` pairs where `key` is a string and `position` is an integer denoting the location of a field within the record. The header record may be accessed by `header`.

Examples

```
> using csv;
> using namespace csv;
> let d = dialect {delimiter=>"\t"};
> let f = open ("junk.csv", "w", d);
> putr f {"hello",123,"",3+:4,world};
()
> close f;
()
> let f = open ("junk.csv", "r", d);
> getr f;
```

```
{"hello","123","","3+:4","world"}
>
```

Suppose our file "test.csv" is as presented below.

```
ir$ more test.csv
NAME,TEST1,TEST2
"HOPE, BOB",90,95
"JONES, SALLY",88,72
"RED, FEEFEE",45,52
```

Notice how the `LIST` option affects the return of `getr` and how the `HEADER` option may be used to index records.

```
> using csv;
> using namespace csv;
> let d = dialect {quote_flag=>MINIMAL};
> let f = open ("test.csv", "r", d, [LIST,HEADER]);
> let r = getr f;
> r!0;
"HOPE, BOB"
> let k = header f;
> k;
{"NAME"=>0,"TEST1"=>1,"TEST2"=>2}
> r!(k!"NAME");
"HOPE, BOB"
> r!!(k!!["NAME","TEST1"]);
["HOPE, BOB",90]
>
```

19.2.4 File Reading Functions

`header csv_file::pointer` returns the record of `key=>position` pairs when opened by `csv::open` using the header option. If the file was opened without the `HEADER` option, `{}` is returned.

`getr csv_file::pointer` reads from a `csv_file` opened by `csv::open` and returns a record represented as a row matrix. Reading from a file opened for writing or appending invokes the `error msg` rule.

`fgetr csv_file::pointer` reads a whole file and returns a list of records. This procedure should only be used on data files that are small enough to fit in the computer's primary memory. Reading from a file opened for writing or appending invokes the `error msg` rule.

`fgetr_lazy csv_file::pointer` Lazy version of `fgetr`.

19.2.5 File Writing Functions

When modifying CSV files that will be imported into Microsoft Excel, fields with significant leading 0s should be written using a `"=""0..."""` formatting scheme. This same technique will work for preserving leading space too. Again, this quirk should only be necessary for files to be imported into MS Excel.

`putr csv_file::pointer rec::matrix` writes a record in row matrix format to `csv_file`. Writing to a file opened for reading invokes the `error msg` rule.

`fputr csv_file::pointer l@(_:_)` writes a list of records where each record is a row matrix to `csv_file`. Writing to a file opened for reading invokes the `error msg` rule.

19.2.6 Examples

The first example shows how to write and read a default CSV file.

```
> using csv;
> using namespace csv;
> let f = open ("testing.csv", "w");
> fputr f [{"bob",3.9,"",-2},{"fred",-11.8,"",0},{"mary",2.3,"$",11}];
()
> close f;
()
> let f = open "testing.csv";
> fgetr f;
[{"bob","3.9","","-2"},{"fred","-11.8","","0"},{"mary","2.3","$","11"}]
> close f;
>
```

The second example illustrates how to write and read a CSV file using automatic conversions.

```
> using csv;
> using namespace csv;
> let d = dialect {quote_flag=>MINIMAL};
> let f = open ("test.csv", "w", d);
> putr f {"I","",-4,1.2,2%4,like};
()
> putr f {"playing","the",0,-0.2,1+:4,drums};
()
> close f;
()
> let f = open ("test.csv", "r", d);
> fgetr f;
[{"I","",-4,1.2,"2%4","like"},{"playing","the",0,-0.2,"1+:4","drums"}]
> close f;
()
>
```

Records containing quotes, delimiters, and line breaks are also properly handled.

```
> using csv;
> using namespace csv;
> let d = dialect {quote_flag=>STRING};
> let f = open ("test.csv", "w", d);
> fputr f [{"this\nis\n",1},{"a \"test\"",2}];
()
> close f;
()
> let f = open ("test.csv", "r", d);
> fgetr f;
[{"this\nis\n",1},{"a \"test\"",2}]
> close f;
()
>
```

Consider the following hand written CSV file. According to RFC4180, this is not a valid CSV file. However, by using the `space_around_quoted_field`, the file can still be read.

```
erucker:$ more test.csv
  "this",   "is",  "not", "valid"

> using csv;
> using namespace csv;
> let f = open "test.csv";
> getr f;
csv::error "parse error at line 1"
> let d = dialect {space_around_quoted_field=>BOTH};
> let f = open ("test.csv", "r", d);
> getr f;
{"this","is","not","valid"}
>
```

The `trim_space` flag should be used with caution. A field with space in front of a number should be interpreted as a string, but consider the following file.

```
erucker:$ more test.csv
"  this   ", 45 ,23,  hello
```

Now observe the differences for the two dialects below.

```
> using csv;
> using namespace csv;
> let d = dialect {trim_space=>BOTH};
> let f = open ("test.csv","r",d);
> getr f;
{"this","45","23","hello"}
> let d = dialect {trim_space=>BOTH, quote_flag=>MINIMAL};
> let f = open ("test.csv", "r", d);
> getr f;
{"this",45,23,"hello"}
>
```

The `trim_space` flag also affects writing.

```
> using csv;
> using namespace csv;
> let d = dialect {trim_space=>BOTH};
> let f = open ("test.csv", "w", d);
> putr f {"   this   ","   45 "};
()
> close f;
()
> quit

erucker:$ more test.csv
"this","45"
```

For the last example a tab delimiter is used, automatic conversions is on, and records are represented as lists. Files are automatically closed when the script is finished.

```
> using csv;
> using namespace csv;
> let d = dialect {quote_flag=>MINIMAL, delimiter=>"\t"};
> let f = open ("test.csv", "w", d, [LIST]);
> fputr f [["a","b",-4.5,""],["c","d",2.3,"-"]];
()
> close f;
()
> let f = open ("test.csv", "r", d, [LIST]);
> fgetr f;
[["a","b",-4.5,""],["c","d",2.3,"-"]]
> quit
```

Chapter 20

pure-fastcgi: FastCGI module for Pure

Version 0.5, June 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

This module lets you write FastCGI scripts with Pure, to be run by web servers like Apache. Compared to normal CGI scripts, this has the advantage that the script keeps running and may process as many requests from the web server as you like, in order to reduce startup times and enable caching techniques. Most ordinary CGI scripts can be converted to use FastCGI with minimal changes.

20.1 Copying

Copyright (c) 2009 by Albert Graef. pure-fastcgi is distributed under a 3-clause BSD-style license, please see the included COPYING file for details.

20.2 Installation

Besides Pure, you'll need to have the FastCGI library installed to compile this module. Also, to run FastCGI scripts, your web server must be configured accordingly; see the documentation of FastCGI and your web server for details.

Get the latest source from http://pure-lang.googlecode.com/files/pure-fastcgi-0.5.tar.gz.

Running `make` compiles the module, `make install` installs it in your Pure library directory. You might have to adjust the path to the fcgi_stdio.h header file in fastcgi.c and/or the option to link in the FastCGI library in the Makefile.

The Makefile tries to guess the host system type and Pure version, and set up some platform-specific things accordingly. If this doesn't work for your system then you'll have to edit the Makefile accordingly.

20.3 Usage

pure-fastcgi provides the `accept` function with which you tell the FastCGI server that your script is ready to accept another request.

`fastcgi::`**`accept`**
: Accept a FastCGI request.

The module also overrides a number of standard I/O functions so that they talk to the server instead. These routines are all in the `fastcgi` namespace. In your Pure script, you can set up a simple loop to process requests as follows:

```
#!/usr/local/bin/pure -x

using fastcgi;
using namespace fastcgi;

extern char *getenv(char*);

main count = main count when
  count = count+1;
  printf "Content-type: text/html\n\n\
<title>FastCGI Hello! (Pure, fcgi_stdio library)</title>\
<h1>FastCGI Hello! (Pure, fcgi_stdio library)</h1>\
Request number %d running on host <i>%s</i>\n"
  (count,getenv "SERVER_NAME");
end if accept >= 0;

main 0;
```

(You might have to adjust the "shebang" in the first line above, so that the shell finds the Pure interpreter. Also, remember to make the script executable. If you're worried about startup times, or if your operating system doesn't support shebangs, then you can also use the Pure interpreter to compile the script to a native executable instead.)

This script keeps running until `accept` returns `-1` to indicate that the script should exit. Each call to `accept` blocks until either a request is available or the FastCGI server detects an error or other kind of termination condition. As with ordinary CGI, additional information about the request is available through various environment variables. A list of commonly supported environment variables and their meaning can be found in The Common Gateway Interface specification.

A number of other routines are provided to deal with data filters, finish a request and set an exit status for a request. These correspond to operations provided by the FastCGI library, see the FastCGI documentation and the FCGI_Accept(3), FCGI_StartFilterData(3), FCGI_Finish(3) and FCGI_SetExitStatus(3) manpages for details. An interface to the FCGI_ToFILE macro is also available. Note that in Pure these functions are called `accept`, `start_filter_data`, `finish`, `set_exit_status` and `to_file`, respectively, and are all declared in the `fastcgi` namespace. A detailed listing of all routines can be found in the fastcgi.pure module.

Please see the examples subdirectory in the pure-fastcgi sources for some more elaborate examples.

Note that to run your FastCGI scripts in a browser, your web server must have the FastCGI module loaded and must also be set up to execute the scripts. E.g., when using Apache, the following configuration file entry will set up a directory for FastCGI scripts:

```
ScriptAlias /fastcgi-bin/ "/srv/www/fastcgi-bin/"
<Location /fastcgi-bin/>
    Options ExecCGI
    SetHandler fastcgi-script
    Order allow,deny
    Allow from all
</Location>
```

(Replace `fastcgi-script` with `fcgid-script` if you're running `mod_fcgid` rather than `mod_fastcgi`.)

Put this entry into http.conf or a similar file provided by your Apache installation (usually under /etc/apache2), and restart Apache. After that you can just throw your scripts into the `fastcgi-bin` directory to have them executed via an URL like `http://localhost/fastcgi-bin/myscript`.

You can also set up a handler for the `.pure` filename extension as follows:

```
<IfModule mod_fastcgi.c>
<FilesMatch "\.pure$">
    AddHandler fastcgi-script .pure
    Options +ExecCGI
</FilesMatch>
</IfModule>
```

(Again, you'll have to adjust the `IfModule` statement and replace `fastcgi-script` with `fcgid-script` if you're running `mod_fcgid`.) After that you should be able to execute scripts with the proper extension anywhere under your server's document root.

Chapter 21

Pure-ODBC - ODBC interface for the Pure programming language

Version 0.9, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>
Jiri Spitz <jiri.spitz@bluetone.cz>

This module provides a simple ODBC interface for the Pure programming language, which lets you access a large variety of open source and commercial database systems from Pure. ODBC a.k.a. "Open Database Connectivity" was originally developed by Microsoft for Windows, but is now available on many different platforms, and two open source implementations exist for Unix-like systems: iODBC (http://www.iodbc.org) and unixODBC (http://www.unixodbc.org).

ODBC has become the industry standard for portable and vendor independent database access. Most modern relational databases provide an ODBC interface so that they can be used with this module. This includes the popular open source DBMSs MySQL (http://www.mysql.com) and PostgreSQL (http://www.postgresql.org). The module provides the necessary operations to connect to an ODBC data source and retrieve or modify data using SQL statements.

To make this module work, you must have an ODBC installation on your system, as well as the driver backend for the DBMS you want to use (and, of course, the DBMS itself). You also have to configure the DBMS as a data source for the ODBC system. On Windows this is done with the ODBC applet in the system control panel. For iODBC and unixODBC you can either edit the corresponding configuration files (/etc/odbc.ini and/or ~/.odbc.ini) by hand, or use one of the available graphical setup tools. More information about the setup process can be found on the iODBC and unixODBC websites.

21.1 Copying

Copyright (c) 2009 by Albert Graef <Dr.Graef@t-online.de>.
Copyright (c) 2009 by Jiri Spitz <jiri.spitz@bluetone.cz>.

pure-odbc is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

21.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-odbc-0.9.tar.gz.

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure installed. The only other dependency is the GNU Multiprecision Library (GMP).

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. The variable `ODBCLIB` specifies the ODBC library to be linked with. The default value is `ODBCLIB=-lodbc`. Please see the Makefile for details.

21.3 Opening and Closing a Data Source

To open an ODBC connection, you have to specify a "connect string" which names the data source to be used with the `odbc::connect` function. A list of available data sources can be obtained with the `odbc::sources` function. For instance, on my Linux system running MySQL and PostgreSQL it shows the following:

```
> odbc::sources;
[("myodbc","MySQL ODBC 2.50"),("psqlodbc","PostgreSQL ODBC")]
```

The first component in each entry of the list is the name of the data source, which can be used as the value of the `DSN` option in the connect string, the second component provides a short description of the data source.

Likewise, the list of ODBC drivers available on your system can be obtained with the `odbc::drivers` function which returns a list of pairs of driver names and attributes. (Older ODBC implementations on Unix lacked this feature, but it seems to be properly supported in recent unixODBC implementations at least.) This function can be used to determine a legal value for the `DRIVER` attribute in the connect string, see below.

The `odbc::connect` function is invoked with a single parameter, the connect string, which is used to describe the data source and various other parameters such as user id and password. For instance, on my system I can connect to the local `myodbc` data source from above as follows:

```
> let db = odbc::connect "DSN=myodbc";
```

The `odbc::connect` function returns a pointer to an `ODBCHandle` object which is used to refer to the database connection in the other routines provided by this module. An `ODBCHandle` object is closed automatically when it is no longer accessible. You can also close it explicitly with a call to the `odbc::disconnect` function:

```
> odbc::disconnect db;
```

After `odbc::disconnect` has been invoked on a handle, any further operations on it will fail.

`odbc::connect` allows a number of attributes to be passed to the ODBC driver when opening the database connection. E.g., here's how to specify a username and password; note that the different attributes are separated with a semicolon:

```
> let db = odbc::connect "DSN=myodbc;UID=root;PWD=guess";
```

The precise set of attributes in the connect string depends on your ODBC driver, but at least the following options should be available on most systems. (Case is insignificant in the attribute names, so e.g. the `DATABASE` attribute may be specified as either `DATABASE`, `Database` or `database`.)

- DSN=<data source name>
- DRIVER=<driver name>
- HOST=<server host name>
- DATABASE=<database path>
- UID=<user name>
- PWD=<password>

The following attributes appear to be Windows-specific:

- FILEDSN=<DSN file name>
- DBQ=<database file name>

Using the `FILEDSN` option you can establish a connection to a data source described in a .dsn file on Windows, as follows:

```
> odbc::connect "FILEDSN=test.dsn";
```

Usually it is also possible to directly connect to a driver and name a database file as the data source. For instance, using the MS Access ODBC driver you can connect to a database file test.mdb as follows:

```
> odbc::connect "DRIVER=Microsoft Access Driver (*.mdb);DBQ=test.mdb";
```

SQLite (http://www.sqlite.org) provides another way to get a database up and running quickly. For that you need the SQLite library and the SQLite ODBC driver available at http://www.ch-werner.de/sqliteodbc. Then you can open an SQLite database as follows (the database file is named with the `DATABASE` attribute and is created automatically if it doesn't exist):

```
> odbc::connect "DRIVER=SQLite3;Database=test.db";
```

SQLite generally performs very well if you avoid some pitfalls (in particular, big batches of updates/inserts should be done within a transaction, otherwise they will take forever). It is certainly good enough for smaller databases and very easy to set up. Basically, after installing SQLite and its ODBC driver you're ready to go immediately. This makes it a very convenient alternative if you don't want to go through the tedium of setting up one of the big hulking DBMS.

21.4 Getting Information about a Data Source

You can get general information about an open database connection with the `odbc::info` function. This function returns a tuple of strings with the following items (see the description of the SQLGetInfo() function in the ODBC API reference for more information):

- DATA_SOURCE_NAME: the data source name
- DATABASE_NAME: the default database
- DBMS_NAME: the host DBMS name
- DBMS_VER: the host DBMS version
- DRIVER_NAME: the name of the ODBC driver
- DRIVER_VER: the version of the ODBC driver
- DRIVER_ODBC_VER: the ODBC version supported by the driver
- ODBC_VER: the ODBC version of the driver manager

E.g., here is what the connection to MySQL shows on my Linux system:

```
> odbc::info db;
"myodbc","test","MySQL","5.0.18","myodbc3.dll","03.51.12","03.51","03.52"
```

The odbc module also provides a number of operations to retrieve a bunch of additional meta information about the given database connection. In particular, the `odbc::getinfo`

function provides a direct interface to the SQLGetInfo() routine. The result of `odbc::getinfo` is a pointer which can be converted to an integer or string value, depending on the type of information requested. For instance:

```
> get_short $ odbc::getinfo db odbc::SQL_MAX_TABLES_IN_SELECT;
31

> cstring_dup $ odbc::getinfo db odbc::SQL_IDENTIFIER_QUOTE_CHAR;
"'"
```

Information about supported SQL data types is available with the `odbc::typeinfo` routine (this returns a lot of data, see odbc.pure for an explanation):

```
> odbc::typeinfo db odbc::SQL_ALL_TYPES;
```

Moreover, information about the tables in the current database, as well as the structure of the tables and their primary and foreign keys can be retrieved with the `odbc::tables`, `odbc::columns`, `odbc::primary_keys` and `odbc::foreign_keys` functions:

```
> odbc::tables db;
[("event","TABLE"),("pet","TABLE")]

> odbc::columns db "pet";
[("name","varchar","NO","''"),("owner","varchar","YES",odbc::SQLNULL),
("species","varchar","YES",odbc::SQLNULL),("sex","char","YES",odbc::SQLNULL),
("birth","date","YES",odbc::SQLNULL),("death","date","YES",odbc::SQLNULL)]

> odbc::primary_keys db "pet";
["name"]

> odbc::foreign_keys db "event";
[("name","pet","name")]
```

This often provides a convenient and portable means to retrieve basic information about table structures, at least on RDBMS which properly implement the corresponding ODBC calls. Also note that while this information is also available through special system catalogs in most databases, the details of accessing these vary a lot among implementations.

21.5 Executing SQL Queries

As soon as a database connection has been opened, you can execute SQL queries on it using the `sql` function which executes a query and collects the results in a list. Note that SQL queries generally come in two different flavours: queries returning data (so-called *result sets*), and statements modifying the data (which have as their result the number of affected rows). The `sql` function returns a nonempty list of lists (where the first list denotes the column titles, and each subsequent list corresponds to a single row of the result set) in the former, and the row count in the latter case.

For instance, here is how you can select some entries from a table. (The following examples assume the sample "menagerie" tables from the MySQL documentation. The `initdb` func-

tion in the examples/menagerie.pure script can be used to create these tables in your default database.)

```
> odbc::sql db "select name,species from pet where owner='Harold'" [];
[["name","species"],["Fluffy","cat"],["Buffy","dog"]]
```

Often the third parameter of `sql`, as above, is just the empty list, indicating a parameterless query. Queries involving marked input parameters can be executed by specifying the parameter values in the third argument of the `sql` call. For instance:

```
> odbc::sql db "select name,species from pet where owner=?" ["Harold"];
[["name","species"],["Fluffy","cat"],["Buffy","dog"]]
```

Multiple parameters are specified as a list:

```
> odbc::sql db "select name,species from pet where owner=? and species=?"
> ["Harold","cat"];
[["name","species"],["Fluffy","cat"]]
```

Parameterized queries are particularly useful for the purpose of inserting data into a table:

```
> odbc::sql db "insert into pet values (?,?,?,?,?,?)"
> ["Puffball","Diane","hamster","f","1999-03-30",odbc::SQLNULL];
1
```

In this case we could also have hard-coded the data to be inserted right into the SQL statement, but a parameterized query like the one above can easily be applied to a whole collection of data rows, e.g., as follows:

```
> do (odbc::sql db "insert into pet values (?,?,?,?,?,?)") data;
```

Parameterized queries also let you insert data which cannot be specified easily inside an SQL query, such as long strings or binary data.

The following SQL types of result and parameter values are recognized and converted to/from the corresponding Pure types:

SQL value/type	Pure value/type
SQL NULL (no value)	`odbc::SQLNULL`
integer types (INTEGER and friends)	`int`
64-bit integers	`bigint`
floating point types (REAL, FLOAT and friends)	`double`
binary data (BINARY, BLOB, etc.)	`(size, data)`
character strings (CHAR, VARCHAR, TEXT, etc.)	`string`

Note the special constant (nonfix symbol) `odbc::SQLNULL` which is used to represent SQL NULL values.

Also note that binary data is specified as a pair `(size, data)` consisting of an int or bigint `size` which denotes the size of the data in bytes, and a pointer `data` (which must not be a null pointer unless `size` is 0 as well) pointing to the binary data itself.

All other SQL data (including, e.g., TIME, DATE and TIMESTAMP) is represented in Pure using its character representation, encoded as a Pure string.

Some databases also allow special types of queries (e.g., "batch" queries consisting of multiple SQL statements) which may return multiple result sets and/or row counts. The `sql` function only returns the first result set, which is appropriate in most cases. If you need to determine all result sets returned by a query, the `msql` function must be used. This function is invoked in exactly the same way as the `sql` function, but returns a list with all the result sets and/or row counts of the query.

Example:

```
> odbc::msql db "select * from pet; select * from event" [];
```

This will return a list with two result sets, one for each query.

21.6 Low-Level Operations

The `sql` and `msql` operations are in fact just ordinary Pure functions which are implemented in terms of the low-level operations `sql_exec`, `sql_fetch`, `sql_more` and `sql_close`. You can also invoke these functions directly if necessary. The `sql_exec` function starts executing a query and returns either a row count or the column names of the first result set as a tuple of strings. After that you can use `sql_fetch` to obtain the results in the set one by one. When all rows have been delivered, `sql_fetch` fails. The `sql_more` function can then be used to check for additional result sets. If there are further results, `sql_more` returns either the next row count, or a tuple of column names, after which you can invoke `sql_fetch` again to obtain the data rows in the second set, etc. When the last result set has been processed, `sql_more` fails.

Example:

```
> odbc::sql_exec db "select name,species from pet where owner='Harold'" [];
["name","species"]
> odbc::sql_fetch db; // get the 1st row
["Fluffy","cat"]
> odbc::sql_fetch db; // get the 2nd row
["Buffy","dog"]
> odbc::sql_fetch db; // no more results
odbc::sql_fetch #<pointer 0x24753e0>
> odbc::sql_more db; // no more result sets
odbc::sql_more #<pointer 0x24753e0>
```

Moreover, the `sql_close` function can be called at any time to terminate an SQL query, after which subsequent calls to `sql_fetch` and `sql_more` will fail:

```
> odbc::sql_close db; // terminate query
()
```

This is not strictly necessary (it will be done automatically as soon as the next SQL query is invoked), but it is useful in order to release all resources associated with the query, such as

parameter values which have to be cached so that they remain accessible to the SQL server. Since these parameters in some cases may use a lot of memory it is better to call `sql_close` as soon as you are finished with a query. This is also done automatically by the `sql` and `msql` functions.

Also note that only a single query can be in progress per database connection at any one time. That is, if you invoke `sql_exec` to initiate a new query, a previous query will be terminated automatically. (However, it is possible to execute multiple queries on the same database simultaneously, if you process them through different connections to that database.)

The low-level operations are useful when you have to deal with large result sets where you want to avoid to build the complete list of results in main memory. Instead, these functions allow you to process the individual elements immediately as they are delivered by the `sql_fetch` function. (An alternative method which combines the space efficiency of immediate processing with the convenience of the list representation is discussed in the following section.) Using the low-level operations you can also build your own specialized query engines; take the definitions of `sql` or `msql` as a start and change them according to your needs.

21.7 Lazy Processing

As an experimental feature, the odbc module also provides two operations `odbc::lsql` and `odbc::lmsql` which work like `odbc::sql` and `odbc::msql` (see Executing SQL Queries above), but return lazy lists (streams) instead. This offers the convenience of a list-based representation without the overhead of keeping entire result sets in memory, which can be prohibitive when working with large amounts of data.

These functions are invoked just like `odbc::sql` and `odbc::msql`, but they return a lazy list of rows (or a lazy list of lazy lists of rows in the case of `lmsql`). For instance:

```
> odbc::lsql db "select * from pet" [];
["name","owner","species","sex","birth","death"]:#<thunk 0x7ffbb9aa2eb8>
```

Note that the tail of the result list is "thunked" and will only be produced on demand, as you traverse the list. As a simple example, suppose that we just want to print the `name` field of each data row:

```
> using system;
> do (\(name:_)->puts name) $ tail $ odbc::lsql db "select * from pet" [];
Fluffy
Claws
Buffy
Fang
Bowser
Chirpy
Whistler
Slim
()
```

Here only one row is in memory at any time while the `do` function is in progress. This

keeps memory requirements much lower than when using the `odbc::sql` function which first loads the entire result set into memory. Another advantage is that only those data rows are fetched from the database which are actually needed in the course of the computation. This can speed up the processing significantly if only a part of the result set is needed. For instance, in the following example we only look at the first two data rows until the desired row is found, so the remaining rows are never fetched from the database:

```
> head [row | row@(name:_) = tail $ odbc::lsql db "select * from pet" [];
>             name == "Claws"];
["Claws","Gwen","cat","m","1994-03-17",odbc::SQLNULL]
```

On the other hand, `lsql/lmsql` will usually be somewhat slower than `sql/msql` if the entire result set is being processed. So you should always consider the time/space tradeoffs when deciding which functions to use in a given situation.

Also note that when using `lsql/lmsql`, the query remains in progress as long as the result list is still being processed. (This is different from `sql/msql` which load the complete result set(s) at once after which the query is terminated immediately.) Since only one query can be executed per database connection, this means that only one lazy result set can be processed per database connection at any time. However, as with the lowlevel operations it is possible to do several lazy queries simultaneously if you assign them to different database connections.

21.8 Error Handling

When one of the above operations fails because the SQL server reports an error, an error term of the form `odbc::error msg state` will be returned, which specifies an error message and the corresponding SQL state (i.e., error code). A detailed explanation of the state codes can be found in the ODBC documentation. For instance, a reference to a non-existent table will cause a report like the following:

```
> odbc::sql db "select * from pets" [];
odbc::error "[TCX][MyODBC]Table 'test.pets' doesn't exist" "S1000"
```

You can check for such return values and take some appropriate action. By redefining odbc::error accordingly, you can also have it generate exceptions or print an error message. For instance:

```
odbc::error msg state = fprintf stderr "%s (%s)\n" (msg,state) $$ ();
```

Note: When redefining `odbc::error` in this manner, you should be aware that the return value of `odbc::error` is what will be returned by the other operations of this module in case of an error condition. These return values are checked by other functions such as `sql`. Thus the return value should still indicate that an error has happened, and not be something that might be interpreted as a legal return value, such as an integer or a nonempty tuple. It is usually safe to have `odbc::error` return an empty tuple or throw an exception, but other types of return values should be avoided.

21.9 Caveats and Bugs

Be warned that multiple result sets are not supported by all databases. I also found that some ODBC drivers do not properly implement this feature, even though the database supports it. So you better stay away from this if you want your application to be portable. You can easily implement batched queries using a sequence of single queries instead.

Note that since the exact numeric SQL data types (NUMERIC, DECIMAL) are mapped to Pure double values (which are double precision floating point numbers), there might be a loss of precision in extreme cases. If this is a problem you should explicitly convert these values to strings in your query, which can be done using the SQL CAST function, as in `select cast(1234.56 as char)`.

21.10 Further Information and Examples

For further details about the operations provided by this module please see the odbc.pure file. A sample script illustrating the usage of the module can be found in the examples directory.

Chapter 22

Pure-Sql3

Version 0.4, June 26, 2012

Peter Summerland <p.summerland@gmail.com>
Albert Graef <Dr.Graef@t-online.de>

This document describes **Sql3**, a SQLite module for the Pure programming language.

22.1 Introduction

SQLite is a software library that implements an easy to use, self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is not intended to be an enterprise database engine like Oracle or PostgreSQL. Instead, SQLite strives to be small, fast, reliable, and above all simple. See Appropriate Uses For SQLite.

Sql3 is a wrapper around SQLite's C interface that provides Pure programmers access to almost all of SQLite's features, including many that are not available through Pure's generic ODBC interface.

22.1.1 Simple Example

Here is a simple example that opens a database file "readme.db" (creating it if it does not exist), adds a table "RM", populates "RM" and executes a query.

```
pure-sql3$> pure -q
>

> using sql3; using namespace sql3;
```

```
> let dbp = open "readme.db";

> exec dbp "create table if not exists RM (name text, age integer)";

> exec dbp "delete from RM";

> let sp1 = prep dbp "ci" "insert into RM values (?,?)";

> exec sp1 ("Sam",20);

> exec sp1 ("Fred",22);

> let sp2 = prep dbp "ci:i" "select * from RM where age > ?";

> exec sp2 18;
[["Sam",20],["Fred",22]]
```

The Sql3 functions, `open`, `prep` and `exec` encapsulate the core functionality of SQLite, and in many cases are all you need to use SQLite effectively.

22.1.2 More Examples

The examples subdirectory of pure-Sql3 contains several files that further illustrate basic usage as well as some of Sql3's more sophisticated features. These include readme.pure, a short file that contains the examples included herein. If you are using emacs pure-mode you can load readme.pure into a buffer and execute the examples line by line (pressing C-c C-c) (as well as experiment as you go).

22.1.3 SQLite Documentation

SQLite's home page provides excellent documentation regarding its SQL dialect as well as its C interface. Comments in this document regarding SQLite are not meant to be a substitute for the actual documentation and should not be relied upon, other than as general observations which may or may not be accurate. The best way to use Sql3 is to get familiar with SQLite and its C interface and go directly to the SQLite Site Map for authoritative answers to any specific questions that you might have.

In the rest of this document, it is assumed the reader has some familiarity with SQLite and has read An Introduction To The SQLite C/C++ Interface.

22.1.4 Sqlite3 - The SQLite Command-Line Utility

The SQLite library includes a really nice command-line utility named sqlite3 (or sqlite3.exe on Windows) that allows the user to manually enter and execute SQL statements against a SQLite database (and much more).

This tool is an invaluable aid when working with SQLite in general and with Sql3 in the Pure interpreter in particular. For example, after entering the Pure statements from the Simple Example above, you could start a new terminal, cd to pure-sql3, type "sqlite3 readme.db" at the prompt, and see the effect the Pure statements had on the database:

```
pure-sql3$> sqlite3 readme.db
SQLite version 3.6.16
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

sqlite> select * from RM;
Sam|20
Fred|22
```

For bottom up REPL development, sqlite3 and Pure are an excellent combination.

22.2 Copying

Copyright (c) 2010 by Peter Summerland <p.summerland@gmail.com>.
Copyright (c) 2010 by Albert Graef <Dr.Graef@t-online.de>.

All rights reserved.

Sql3 is free software: you can redistribute it and/or modify it under the terms of the New BSD License, often referred to as the 3 clause BSD license. Sql3 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Please see the COPYING file for the actual license applicable to Sql3.

22.3 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-sql3-0.4.tar.gz.

Unless you already have them on your machine, download SQLite and sqlite3 from the SQLite website and install as indicated. To install Sql3, cd to the pure-sql3 directory, run `make`, and then run `sudo make install` (on Linux).

22.4 Data Structure

From a client's perspective, the most important of SQLite's data structures are the database connection object "sqlite3" and the prepared statement object "sqlite3_stmt". These are opaque data structures that are made available to users of SQLite's C interface via pointers, sqlite3* and sqlite3_stmt*. At appropriate times, Sql3 creates "cooked" versions of these

pointers that can be used (with care) to call native C functions exposed by SQLite's C interface.

Sql3 introduces two new data types, "db_ptr" and "stmt_ptr" which refer to the cooked versions of sqlite3* and sqlite3_stmt*, respectively. These two new data types are defined using :func: **type**, and therefore can be used as type tags in rule patterns or as the first parameter passed to in the typep function. It follows that all db_ptrs are sqlite3* pointers and all stmt_ptrs are sqlite3_stmt* pointers. Thus, using dbp and sp1 from the introductory example:

```
> typep db_ptr dbp, pointer_type dbp;
1, "sqlite3*"

> typep stmt_ptr sp1, pointer_type sp1;
1, "sqlite3_stmt*"
```

The converse, of course, is not true, as SQLite knows nothing about Sql3, and db_ptrs and stmt_ptrs carry other information in addtion to the underlying pointers provided to them by SQLite.

22.5 Core Database Operations

The core database operations are (a) opening and closing database connections and (b) preparing, executing and closing prepared statements.

22.5.1 Database Connections

Generally speaking, the first step in accessing a database is to obtain a db_ptr that references a database connection object. Once the db_ptr is obtained, it can be used to construct prepared statements for updating and querying the underlying database. The last step is usually to close the database connection (although this is will be done automatically by Sql3 when the db_ptr goes out of scope).

Opening a Database Connection

In Sql3 `open` constructs a database connection and returns a db_ptr that refers to the connection.

`sql3::`**`open`** (file_path::string [,access_mode::int[,custom_bindings]])
: opens a SQLite database file whose name is given by the file_path argument and returns a db_ptr for the associated database connection object created by SQLite.

Example:

```
>  let dbp2 = open "abc.db"; dbp2;
#<pointer 0x992dff8>
```

If the filename is ":memory:" a private, temporary in-memory database is created for the connection.

The basic access modes are:

- SQLITE_OPEN_READONLY - the database is opened in read-only mode. If the database does not already exist, an error is returned.
- SQLITE_OPEN_READWRITE - the database is opened for reading and writing if possible, or reading only if the file is write protected by the operating system. In either case the database must already exist, otherwise an error is returned.
- SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE - the database is opened for reading and writing, and is creates it if it does not already exist. This is the default value that is used if the flags argument is omitted.
- SQLITE_OPEN - an alias for SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE provided by Sql3.

These flags can be combined with SQLITE_OPEN_NOMUTEX SQLITE_OPEN_FULLMUTEX SQLITE_OPEN_SHAREDCACHE SQLITE_OPEN_PRIVATECACHE to control SQLite's threading and shared cache features. All of these flags are exported by Sql3.

The optional custom_bindings argument allows the user to set up customized binding and fetching behavior for prepared statements associated with the returned db_ptr. (See Custom Binding Types for Prepared Statements)

Failure to Open a Database Connection

If SQLite cannot open the connection, it still returns a pointer to a database connection object that must be closed. In this case, `open` automatically closes the the connection object and then throws an exception. E.g.,:

```
> catch error (open ("RM_zyx.db",SQLITE_OPEN_READONLY));
error (sql3::db_error 14 "unable to open database file [open RM_zyx.db]")
```

Apparently, SQLite does not verify that a file is a valid SQLite database when it opens a connection. However, if the file is corrupted SQLite will return an error when the connection is used.

Testing a db_ptr

You can test any object to see if it is a db_ptr using (typep db_ptr):

typep db_ptr x
: returns 1 if x is a db_ptr returned by open, and 0 if it is not.

You can also determine if a db_ptr's data connection is open.

`sql3::`**`is_open`** dbp::db_ptr
: returns 1 if the database connection referenced by dbp is open.

Closing a Database Connection

When a database connection object is no longer needed, it should be closed so that SQLite can free the associated resources.

`sql3::`**`close`** dbp::db_ptr
: if the database connection referenced by the db_ptr dbp is open, close it using sqlite3_close; otherwise do nothing.

Before calling `sqlite3_close`, `close` finalizes all prepared statements associated with the connection being closed. Sql3 will detect and throw a db_error if an attempt is subsequently made to execute a statement associated with the closed database connection.

```
> let dbp2_sp = prep dbp2 "ci:" "select * from RM";

> exec dbp2_sp ();
[["Sam",20],["Fred",22]]

> close dbp2;

> catch error (exec dbp2_sp);
error (sql3::db_error 0 "Attempt to exec on a closed db_ptr.")
```

If a db_ptr goes out of scope, Sql3 will automatically call `sqlite3_close` to close the referenced database connection, but only if the connection has not already been closed by `close`. Thus, for example, it is not necessary to use a catch statement to ensure that Sqlite3 resources are properly finalized when a db_ptr is passed into code that could throw an exception.

When debugging, this activity can be observed by editing sql3.pure, changing "const SHOW_OPEN_CLOSE = 0;" to "const SHOW_OPEN_CLOSE = 1;" and running sudo make install in the pure-sql3 directory. This will cause a message to be printed whenever a db_ptr or stmt_ptr is created or finalized.

N.B. You should never call the native C interface function''sqlite3_close'' with a db_ptr. If the referenced database connection is closed by such a call, a subsequent call to `close` on this db_ptr (including the call that will automatically occur when the db_ptr goes out of scope) will cause a seg fault.

22.5.2 Prepared Statements

The native SQLite C interface provides five core functions needed to execute a SQL statement.

- sqlite3_prepare_v2
- sqlite3_bind

- sqlite3_step
- sqlite3_column
- sqlite3_finalize

Using the C interface, the basic procedure is to prepare a statement using `sqlite3_prepare_v2`, bind its parameters using `sqlite3_bind`, step it using `sqlite3_step` one or more times until it is done and then finalize it using `sqlite3_finalize`. Each time `sqlite3_step` returns SQLITE_ROW, use `sqlite3_column` to fetch the row's values. Here `sqlite3_bind` and `sqlite3_column` represent families of bind and column functions, rather than actual functions, with one member for each of the basic data types recognized by SQLite. Thus, for example, `sqlite_bind_double` is the function one would use to bind a prepared statement with an argument of type double.

Sql3 encapsulates these procedures in four functions: `prep`, `exec`, `lexec` and `finalize`.

Constructing Prepared Statements

In Sql3 you can use `prep` to construct a prepared statement and obtain a stmt_ptr that refers to it.

`sql3::`**`prep`** dbp::db_ptr binding_string::string sql_statement::string
: constructs a prepared statement object and returns a stmt_ptr that references it. `dbp` must be a db_ptr or the rule will not match. `sql_statement` is the SQL statement that will be executed when the prepared statement is passed to `exec`.

Basically, `prep` just passes `dbp` and `sql_statement` on to `sqlite3_prepare_v2` and returns a sentry guarded version of the sqlite3_stmt* it receives back from `sqlite3_prepare_v2`. SQL statements passed to `prep` (and `sqlite3_prepare_v2`) can have argument placeholders, indicated by "?", "?nnn", ":AAA", etc, in which case the argument placeholders must be bound to values using `sqlite_bind` before the prepared statement is passed to `sqlite3_step`. Hence the `binding_string`, which is used by Sql3 to determine how to bind the prepared statement's argument placeholders, if any. The binding string also tells Sql3 how to fetch values in the `sqlite3_column` phase of the basic prepare, bind, step, fetch, finalize cycle dictated by the SQlite C interface.

In the following two examples, the "c" and "i" in the binding strings indicate that (a) a string and an int will be used to bind `sp1`,(b) an int will be used to bind `sp2` and (c) `sp2`, when executed, will return a result set in the form of a list of sublists each of which contains a string and an int.

```
> let sp1 = prep dbp "ci" "insert into RM values (?,?)";

> let sp2 = prep dbp "ci:i" "select * from RM where age > ?";
```

In general, the characters in the type string before the ":", if any, indicate the types in the result set. Those that occur after the ":", if any, indicate the types of the arguments used to bind the prepared statement object. If the type string does not contain a ":", the characters in the type string, if any, are the types of binding arguments.

Sql3 provides the following set of "core" binding types:

Type	Pure Argument	SQLite Type
b	(int, pointer)	blob
c	string	text (utf8)
d	double	float
i	int	int
k	int or bigint	int64
l	bigint	blob
n	Sql3::SQLNULL	NULL
x	expression	blob
v	variant	variant

The "**b**" or blob type is different from the rest in that the Pure argument is specified as a pair. The first element of the pair indicates the length in bytes of the object to be stored and the second element indicates its location in memory. The "**c**" type stands for string (as in "char*"), "**d**" stands for double and "**i**" stands for int. The "**k**" type stands for "key" and maps Pure ints and bigints (within the range of int64) to int64 values in the database. This type is useful when dealing with SQLite's "integer primary keys" and "rowids" both of which are int64. The "**l**" type, in contrast applies to all bigints (and not to ints) and it maps bigints onto blobs, which are generally meaningless in SQL math expressions. The "**n**" type can only appear on the binding side of a type string. The "**v**" type stands for any of "b", "c", "d", "i" or "n", based on the type of the binding argument. A "v" type will be fetched from SQLite according to the native SQLite column type of the corresponding column. The "**x**" type is used to store and reconstruct Pure expressions as binary objects, using the `val` and `blob` functions provided by the Pure prelude.

Users can define custom binding types and pass them as a third parameter to `open`. The resulting db_ptr can be used with the custom binding types to construct prepared statements using `prep`.

Testing a stmt_ptr

You can determine if a given expression is a stmt_ptr using typep.

`typep` stmt_ptr x
: returns 1 if x is a stmt_ptr, otherwise returns 0.

Executing Prepared Statements

In Sql3, the bind, step, column, step, column ... cycle is encapsulated in the `exec` and `lexec` functions.

`sql3::`**`exec`** sp::stmt_ptr args
: use `args` to bind the prepared statement referenced by `sp`, execute it and return the result set as a list. The first parameter, `sp` must be a valid stmt_ptr or the rule will fail.

The second parameter, `args`, is a tuple or list of arguments whose number and type correspond to the bind parameter types specified in the call to `prep` that produced the first parameter `sp`.

Thus, using sp1 and sp2 defined in the introductory example:

```
> exec sp1 ("Tom",30);  //insert Tom
[]

> exec sp2 19;          //select age > 19
[["Sam",20],["Fred",22],["Tom",30]]
```

An error is thrown if the args do not correspond to the specified types.

```
> catch error (exec sp2 "a");
error (sql3::db_error 0 "\"a\" does not have type int")
```

If a prepared statement does not have any binding parameters, the call to `exec` should use `()` as the binding argument.

```
> let sp3 = prep dbp "c:" "select name from RM";

> exec sp3 ();
[["Sam"],["Fred"],["Tom"]]
```

Extra care is required when executing prepared statements that take a blob argument because it must be a pair. In order to preserve the tuple as a pair, binding arguments that include a blob should passed to exec as a list. If passed as a member of a larger tuple, it will be treated as two arguments due to the nature of tuples.

```
> let blb = (100,ptr);

> (a,blb,c);
a,100,ptr,c

> [a,blb,c];
[a,(100,ptr),c]
```

Thus something like "`exec stpx [a,blb,c]`" would work fine, while "`exec stpx (a,blb,c)`" would produce a Sql3 binding exception.

Executing Lazily

The `exec` function returns result sets as eager lists which can sometimes be inefficient or simply not feasible for large result sets. In such cases it is preferable to use `lexec` instead of `exec`.

`sql3::`**`lexec`** stmp::stmt_ptr args
: same as `exec` except that it returns a lazy list.

Example:

```
> lexec sp2 19;
["Sam",20]:#<thunk 0xb6475ab0>
```

Note that no changes to `sp2` were required. In addition, for most purposes the lazy list returned by `lexec` can be processed by the same code that processed the eager list returned by `exec`.

Executing Directly on a db_ptr

For statements that have no parameters and which do not return results, `exec` can be applied to a db_ptr.

`sql3::exec` dbp::db_ptr sql_statement::string
: constructs a temporary prepared statement using `sql_statement`. The SQL statement cannot contain argument placeholders and cannot be a select statement.

Example:

```
> exec dbp "create table if not exists RM (name varchar, age integer)";
```

Executing Against a Busy Database

SQLite allows multiple processes to concurrently read a single database, but when any process wants to write, it locks the entire database file for the duration of its update.

When the native SQLite C interface function `sqlite3_step` (used by `exec`) tries to access a file that is locked by another process, it treats the database as "busy" and returns the SQLITE_BUSY error code. If this happens in a call to `exec` or `lexec`, a `db_busy` exception will be thrown.

You can adjust SQLite's behavior using `sqlite3_busy_handler` or `sqlite3_busy_timeout`.

If the statement is a COMMIT or occurs outside of an explicit transaction, then you can retry the statement. If the statement is not a COMMIT and occurs within a explicit transaction then you should rollback the transaction before continuing.

Grouping Execution with Transactions

No changes can be made to a SQLite database file except within a transaction. Transactions can be started manually by executing a BEGIN statement (i.e., exec dbp "BEGIN"). Manually started transactions persist until the next COMMIT or ROLLBACK statement is executed. Transactions are also ended if an error occurs before the transaction is manually ended using a COMMIT or ROLLBACK statement. This behavior provides the means make a series of changes "atomically."

By default, SQLite operates in autocommit mode. In autocommit mode, any SQL statement that changes the database (basically, anything other than SELECT) will automatically start a

transaction if one is not already in effect. As opposed to manually started transactions, automatically started transactions are committed as soon as the execution of the related statement completes.

The upshot of this, in Sql3 terms, is that unless a transaction is started manually, the database will be updated each time `exec` is called. For a long series of updates or inserts this a can be very slow. The way to avoid this problem is to manually begin and end transactions manually.

Sql3 provides the following convenience functions all of which simply call `exec` with the appropriate statement. For example `begin dbp` is exactly the same as `exec dbp "BEGIN"`.

`sql3::`**`begin`** dbp::db_ptr

`sql3::`**`begin_exclusive`** dbp::db_ptr

`sql3::`**`begin_immediate`** dbp::db_ptr

`sql3::`**`commit`** dbp::db_ptr

`sql3::`**`rollback`** dbp::db_ptr

`sql3::`**`savepoint`** dbp::db_ptr save_point::string

`sql3::`**`release`** dbp::db_ptr save_point::string

`sql3::`**`rollback_to`** dbp::db_ptr save_point::string

Note that transactions created using :func: **begin** and :func: **commit** do not nest. For nested transactions, use :func: **savepoint** and :func: **release**.

Finalizing Prepared Statements

When a prepared statement is no longer needed it should be finalized so that SQLite can free the associated resources.

`sql3::`**`finalize`** sp::stmt_ptr
: finalize the prepared statement referenced by `sp`, which must be a stmt_ptr previously returned by `prep`.

Often there is no need to call `finalize` for a given stmt_ptr because it will be automatically called when the stmt_ptr goes out of scope.

If the stmt_ptr is associated with a database connection that has been closed (which would have caused an exception to be thrown), an attempt to finalize it, including the automatic finalization can occur when stmt_ptr goes out of scope, will cause an exception to be thrown.

```
> catch error (finalize dbp2_sp);
error (sql3::db_error 0 "finalize: STMT attached to a closed db_ptr.")
```

Multiple calls to `finalize` are fine. In contrast, the corresponding native C interface function, `sqlite3_finalize` will cause a seg fault if called with a pointer to a finalized prepared statement object. This is the main reason why you should never call `sqlite3_finalize` with

a stmt_ptr. If the prepared statement referenced by the stmt_ptr is finalized by such a call, a subsequent call to `finalize` with the stmt_ptr (including the call that will automatically occur when the stmt_ptr goes out of scope) will cause a seg fault.

22.5.3 Exceptions

Sql3 throws two types of exceptions, one for outright errors and one for database "busy" conditions.

constructor `sql3::db_error` ec msg

When a Sql3 function detects an error it throws an exception of the form "`db_error ec msg`" where ec is an error code and msg is the corresponding error message. If ec>0, the error was detected by SQLite itself, and ec and msg are those returned by SQLite. If ec==0, the error was detected by Sql3 and msg is a Sql3 specific description of the error. E.g.,

```
> db_error_handler (db_error ec msg) = ()
> when
>   source = if ec > 0 then "SQLite" else "Sql3";
>   printf "%s db_error: ec %d, %s\n" (source,ec,msg);
> end;
> db_error_handler x = throw x;

> catch db_error_handler (exec dbp "select * from NO_TABLE");
SQLite db_error: ec 1, no such table: NO_TABLE
```

constructor `sql3::db_busy` dbp

Sql3 functions `exec` and `lexec` throw exceptions of the form "`db_busy dbp`", where `dbp` is a db_ptr, if they are prevented from executing successfully because the database referenced by `dbp` is locked (See Executing Against a Busy Database).

SQLite Error Codes

Here is a list, as of January 31, 2011, of SQLite's error codes.

```
SQLITE_ERROR        1   /* SQL error or missing database */
SQLITE_INTERNAL     2   /* Internal logic error in SQLite */
SQLITE_PERM         3   /* Access permission denied */
SQLITE_ABORT        4   /* Callback routine requested an abort */
SQLITE_BUSY         5   /* The database file is locked */
SQLITE_LOCKED       6   /* A table in the database is locked */
SQLITE_NOMEM        7   /* A malloc() failed */
SQLITE_READONLY     8   /* Attempt to write a readonly database */
SQLITE_INTERRUPT    9   /* Operation terminated by sqlite3_interrupt()*/
SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
SQLITE_CORRUPT     11   /* The database disk image is malformed */
SQLITE_NOTFOUND    12   /* NOT USED. Table or record not found */
SQLITE_FULL        13   /* Insertion failed because database is full */
SQLITE_CANTOPEN    14   /* Unable to open the database file */
```

```
SQLITE_PROTOCOL    15   /* NOT USED. Database lock protocol error */
SQLITE_EMPTY       16   /* Database is empty */
SQLITE_SCHEMA      17   /* The database schema changed */
SQLITE_TOOBIG      18   /* String or BLOB exceeds size limit */
SQLITE_CONSTRAINT  19   /* Abort due to constraint violation */
SQLITE_MISMATCH    20   /* Data type mismatch */
SQLITE_MISUSE      21   /* Library used incorrectly */
SQLITE_NOLFS       22   /* Uses OS features not supported on host */
SQLITE_AUTH        23   /* Authorization denied */
SQLITE_FORMAT      24   /* Auxiliary database format error */
SQLITE_RANGE       25   /* 2nd parameter to sqlite3_bind out of range */
SQLITE_NOTADB      26   /* File opened that is not a database file */
```

New error codes may be added in future versions of SQLite. Note that the SQLite names of the error codes are not exported by the Sql3 module.

22.6 Advanced Features

Sql3's advanced features include the ability to implement SQL functions in Pure, convenient access to the SQLite C interface and custom binding types.

22.6.1 Custom SQL Functions

An extremely powerful (albeit complex) feature of the SQLite C interface is the ability to add new SQL scalar or aggregate functions. The new functions can be used in SQL statements the in same way as SQLites's prepackaged functions. Sql3 hides the complexity and seamlessly integrates all of this functionality, :), into Pure via `create_function`. This function is used to register both scalar SQL functions and aggregate SQL functions with SQlite.

Scalar SQL Functions

You can add a custom SQL scalar function to SQLite by passing a single Pure function to `create_function`.

`sql3::`**`create_function`** dbp::db_ptr name::string nargs::int pure_fun
: registers a new SQL scalar function of `nargs` arguments that can be called, as `name`, in SQL statements prepared with respect to `dbp`, a db_ptr. When the SQL function is called in a SQL statement, control is passed to `pure_fun`, a function written in Pure. If `nargs` is `(-1)`, the SQL function `name` is variadic, and the arguments will be passed to `pure_fun` as a single list.

Note that `create_function` can also register aggregate functions (see Aggregate SQL Functions).

Here is an example of a scalar function that takes two parameters. Note that any kind of Pure "function" can be passed here; local functions, global functions, lambdas or partial applications all work.

```
> create_function dbp::dbp "p_fn" 2 plus with plus x y = x + y; end;

> let sp4 = prep dbp "cii:"
>           "select p_fn('Hi ',name), age, p_fn(age,10) from RM";

> exec sp4 ();
[["Hi Sam",20,30],["Hi Fred",22,32]]
```

Here is an example of a variadic function:

```
> create_function dbp "p_qm" (-1) quasimodo with
>   quasimodo xs = "quasimodo: "+join ":" [str x | x=xs];
> end;
```

If the SQL function takes no arguments, the corresponding Pure function must, for technical reasons in Pure, take a single dummy argument. E.g.,

```
> create_function dbp "p_count" 0 counter with
>   counter () = put r (get r+1);
> end when r = ref 0 end;
```

Here is how `count` and `quasimodo` might be used:

```
> let sp5 = prep dbp "ic:" "select p_count(), p_qm(name,age) from RM";

> exec sp5 ();
[[1,"quasimodo: \"Sam\":20"],[2,"quasimodo: \"Fred\":22"]]

> exec sp5 ();
[[3,"quasimodo: \"Sam\":20"],[4,"quasimodo: \"Fred\":22"]]
```

Multiple SQL functions can be registered with the same name if they have differing numbers of arguments. Built-in SQL functions may be overloaded or replaced by new application-defined functions.

Generally, a custom function is permitted to call other Sql3 and native SQLite C interface functions. However, such calls must not close the database connection nor finalize or reset the prepared statement in which the function is running.

Aggregate SQL Functions

You can use `create_function` to register an aggregate SQL function with SQLite by passing a triple consisting of two Pure functions and a start value, in lieu of a single Pure function.

`sql3::`**`create_function`** dbp::db_ptr name::string nargs::int (step,final,start)
: registers a new SQL aggregate function of `nargs` arguments that can be called, as `name` in SQL statements prepared with respect to `dbp`, a db_ptr. `step` and `final` are curried

Pure functions and `start` is the initial value for the aggregation. The `step` function is called repeatedly to accumulate values from the database, starting from the given `start` value, and finally the `final` function is applied to the accumulated result.

Note that for a single-argument step function, this works exactly as if the functions were invoked as "`final (foldl step start values)`", where `values` is the list of aggregated values from the database.

```
> create_function dbp "p_avg" 1 (step,final,(0,0.0)) with
>   step (n,a) x = n+1, a+x;
>   final (n,a) = a/n;
> end;

> let sp6 = prep dbp "id:" "select count(name), p_avg(age) from RM";

> exec sp6 ();
[[2,21.0]]
```

22.6.2 Accessing the Rest of SQLite's C Interface

The db_ptrs returned by `open` and stmt_ptrs returned by `prep` are sentry guarded versions of the actual pointers to the data base connection objects and prepared statement objects returned by their corresponding native C interface functions `sqlite3_open_v2` and `sqlite3_prepare_v2`. This makes it easy to call almost any external function in SQLite's C interface directly, passing it the same db_ptr or stmt_ptr that is passed to Sql3's functions, such as `prep` or `exec`.

For example, you can override SQLite's default behavior with respect to a busy database as follows:

```
> extern int sqlite3_busy_timeout(sqlite3*, int);

> sqlite3_busy_timeout dbp 10;
```

This sets a busy handler that will "sleep and retry" multiple times until at least 10 milliseconds of sleeping have accumulated. Calling this routine with an argument less than or equal to zero turns off all busy handlers.

Another example is to query the number of database rows that were changed, inserted or deleted by the most recently completed SQL statement executed on a given database connection:

```
> extern int sqlite3_changes(sqlite3*);

> exec sp1 ("Harvey",30);

> sqlite3_changes dbp;
1
```

As a final example, in this case using a stmt_ptr, you can determine name assigned to a column in a result using `sqlite3_column_name`:

```
> extern char *sqlite3_column_name(sqlite3_stmt*, int);

> exec sp2 29;
[["Harvey",30]]

> sqlite3_column_name sp2 1;
"age"
```

In order to call a native C function you must first make it accessible using an extern statement.

Please note also that directly calling a function provided by the SQLite C interface can be dangerous, as is the case with any call from Pure code to an external C function. Sql3 users should be especially careful in this regard because using a db_ptr or a stmt_ptr in calls to certain native C interface functions, including in particular `sqlite3_close` and `sqlite3_finalize`, will corrupt data held by the db_ptr or stmt_ptr, leading to undefined behavior. The reason for this restriction is that Sql3 uses sentries to insure that the resources associated with a db_ptr or a stmt_ptr are automatically finalized by SQLite when they go out of scope. In addition, the sentries carry internal information used by Sql3 for other purposes.

22.6.3 Custom Binding Types for Prepared Statements

You can add your own binding types for preparing and executing prepared statements by specifying a third argument to `open`. The third argument must be a list of "hash rocket pairs" in which the left side is a character for the custom binding type and the right side is a list with three members. The second and third members of the list are functions that map objects from the new type to one of the Sql3 core types and back, respectively. The first member of the list is the character for the Sql3 core types referenced by the mapping functions.

The file sql3_user_bind_types.pure in the examples subdirectory shows how this might be done for a couple of user defined types. The example script deals with dates and certain Pure expressions as bigints and native Pure expressions, while the database stores these as utf-8 text. The following snippets show parts of the script that are relevant to this discussion:

```
const custom_binds = [
  "t"=>["c",day_to_str,str_to_day],
  "s"=>["c",str,eval]
];

d1 = str_to_day "2010-03-22";

db = open ("abc.db", SQLITE_OPEN, custom_binds);
stm1 = prep db "cts" "insert into TC values(?,?,?)";
exec stm1 ["Manny", d1, s_expr];
stm3a = sql3::prep db "t:" "select t_date from TC";
stm3b = sql3::prep db "c:" "select t_date from TC";
```

Executing stm3a and stm3b from the interpreter shows that TC's date field is stored as a

string, but returned to the Pure script as a bigint.

```
> sql3::exec stm3a ());
[[14691L]]

> sql3::exec stm3b ());
[["2010-03-22"]]
```

The character designating the custom type must not be one of the letters used to designate Sql3 core binding types.

22.7 Threading Modes

SQLite supports three different threading modes:

1. Single-thread. In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. Multi-thread. In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. Serialized. In serialized mode, SQLite can be safely used by multiple threads with no restriction.

SQLite can be compiled with or without support for multithreading and the default is to support it.

In many cases, single-thread mode might be appropriate if only because it is measurably faster. This might be the case, for example, if you are using SQLite as the on-disk file format for a desktop application.

If your version of SQLite was compiled with support for multithreading, you can switch to single-thread mode at runtime by calling sqlite3_config() with the verb SQLITE_CONFIG_SINGLETHREAD.

If you must use threads, it is anticipated that Sql3 probably will not impose an additional burden. Hopefully, you will be fine if you apply the same precautions to a db_ptr or stmt_ptr that you would apply to the underlying sqlite* and sqlite_stmt*s if you were not using Sql3. It is strongly advised however that you look at the underlying Sql3 code to verify that this will work. Since everything that is imposed between the raw pointers returned by the SQlite interface and the corresponding db_ptr and stmt_ptrs is written in Pure, it should be relatively easy to determine how Sql3 and your multithreading strategy will interact. See Is SQLite threadsafe? , Opening A New Database Connection and Test To See If The Library Is Threadsafe.

Chapter 23

Pure-XML - XML/XSLT interface

Version 0.6, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

XML, the Extensible Markup Language, facilitates the exchange of complex structured data between applications and systems. XSLT allows you to transform XML documents to other XML-based formats such as HTML. Together, XML and XSLT let you create dynamic web content with ease. Both XML and XSLT are open standards by the W3C consortium (http://www.w3.org).

Pure's XML interface is based on the libxml2 and libxslt libraries from the GNOME project. If you have a Linux system then you most likely have these libraries, otherwise you can get them from http://xmlsoft.org. For Windows users, the required dlls are available from the GnuWin32 project (http://gnuwin32.sourceforge.net) and are already included in the Pure MSI package.

23.1 Copying

Copyright (c) 2009 by Albert Graef <Dr.Graef@t-online.de>.

pure-xml is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

23.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-xml-0.6.tar.gz.

Run `make` and then `sudo make install` to compile and install this module. This requires libxml2, libxslt and Pure.

23.3 Usage

Use the following declaration to make the operations of this module available in your programs:

```
using xml;
```

The module defines two namespaces `xml` and `xslt` for the XML and the XSLT operations, respectively. For convenience, you can open these in your program as follows:

```
using namespace xml, xslt;
```

A number of complete examples illustrating the use of this module can be found in the examples directory in the source distribution.

23.4 Data Structure

This module represents XML documents using pointers to the `xmlDoc` and `xmlNode` structures provided by the libxml2 library. Similarly, stylesheets are simply pointers to the xmlStylesheet structure from libxslt (cf. Transformations). This makes it possible to use these objects directly with the operations of the libxml2 and libsxslt libraries (via Pure's C interface) if necessary. Note, however, that these are all "cooked" pointers which take care of freeing themselves automatically when they are no longer needed, therefore you shouldn't free them manually.

You can also check for these types of pointers using the following predicates:

`xml::`**`docp`** x
: checks whether x is an XML document pointer.

`xml::`**`nodep`** x
: checks whether x is a pointer to a node in an XML document.

`xslt::`**`stylesheetp`** x
: checks whether x is an XSLT stylesheet pointer.

23.4.1 The Document Tree

An XML document is a rooted tree which can be created, traversed and manipulated using the operations of this module. There are different types of nodes in the tree, each carrying their own type of data. In Pure land, the node data is described using the following "node info" constructors.

constructor `xml::element` tag ns attrs
: An XML element with given (possibly qualified) name `tag`, a (possibly empty) list of namespace declarations `ns` and a (possibly empty) list of attributes `attrs`. Namespace declarations normally take the form of a pair of strings `(prefix,href)`, where `prefix` is the prefix associated with the namespace and `href` the corresponding URI (the name of the namespace), but they can also be just a string `href` if the namespace prefix is missing. Attributes are encoded as `key=>value` pairs, where `key` is the attribute name and `value` the associated value; both `key` and `value` are strings.

constructor `xml::element_text` tag ns attrs content
: A convenience function which denotes a combination of an element node with a text child. This is only used when creating a new node, and indicates that a text node child is to be added to the node automatically.

constructor `xml::attr` key val
: An attribute node. These only occur as results of the `select` and `attrs` functions, and cannot be inserted directly into a document.

constructor `xml::text` content
: A text node with the given content (a string).

constructor `xml::cdata` content
: Like `xml::text`, but contains unparsed character data.

constructor `xml::comment` content
: A comment.

constructor `xml::entity_ref` name
: An entity reference (`&name;`).

constructor `xml::pi` name content
: Processing instructions. `name` is the application name, `content` the text of the processing instructions.

23.4.2 Document Types

Besides the node types described above, there are some additional node types used in the document type definition (DTD), which can be extracted from a document using the `int_subset` and `ext_subset` functions. These are for inspection purposes only; it is not possible to change the DTD of a document in-place. (However, you can create a new document and attach a DTD to it, using the `new_doc` function.)

constructor `xml::`**`doctype`** name extid
: DTDs are represented using this special type of node, where `name` is the name of the root element, and `extid` is a pair consisting of the external identifier and the URI of the DTD (or just the URI if there is no external identifier). The `xml::doctype` node has as its children zero or more of the following kinds of DTD declaration nodes (these are just straightforward abstract syntax for the !ELEMENT, !ATTLIST and !ENTITY declarations inside a DTD declaration; see the XML specification for details).

Element declarations: Here, `name` is the element tag and `content` the definition of the element structure, see element content below. XML supports various kinds of element types, please refer to document type definition in the XML specification for details.

constructor `xml::`**`undefined_element`** name
: An undefined element. This is in libxml2, but not in the XML specification, you shouldn't see this in normal operation.

constructor `xml::`**`empty_element`** name
: An element without any content.

constructor `xml::`**`any_element`** name
: An element with unrestricted content.

constructor `xml::`**`mixed_element`** name content
: A "mixed" element which can contain character data, optionally interspersed with child elements, as given in the `content` specification.

constructor `xml::`**`std_element`** name content
: A standard element consisting *only* of child elements, as given in the `content` specification.

Attribute declarations: These are used to declare the attributes of an element. `elem_name` is the name of an element which describes the attribute type, `name` is the name of the attribute itself, and `default` specifies the default value of the attribute, see attribute defaults below. XML supports a bunch of different attribute types, please refer to document type definition in the XML specification for details.

constructor `xml::`**`cdata_attr`** elem_name name default

constructor `xml::`**`id_attr`** elem_name name default

constructor `xml::`**`idref_attr`** elem_name name default

constructor `xml::`**`idrefs_attr`** elem_name name default

constructor `xml::`**`entity_attr`** elem_name name default

constructor `xml::`**`entities_attr`** elem_name name default

constructor `xml::`**`nmtoken_attr`** elem_name name default

constructor `xml::`**`nmtokens_attr`** elem_name name default

constructor `xml::`**`enum_attr`** elem_name name vals default

constructor `xml::`**`notation_attr`** elem_name name vals default

Entity declarations: These are used for internal and external entity declarations. `name` is the entity name and `content` its definition. External entities also have an `extid` (external identifier/URI pair) identifying the entity.

constructor `xml::`**`int_entity`** name content

constructor `xml::`**`int_param_entity`** name content

constructor `xml::`**`ext_entity`** name extid content

constructor `xml::`**`ext_param_entity`** name extid content

The element content type (`content` argument of the element declaration nodes) is a kind of regular expression formed with tags (specified as strings) and the following constructors:

constructor `xml::`**`pcdata:`** text data (''#PCDATA'')

constructor `xml::`**`sequence`** xs: concatenation (''x,y,z'')

constructor `xml::`**`union`** xs: alternatives (''x | y | z'')

constructor `xml::`**`opt`** x: optional element (''x?'')

constructor `xml::`**`mult`** x: repeated element (''x*'')

constructor `xml::`**`plus`** x: non-optional repeated element (''x+'')

Attribute defaults (the `default` argument of attribute declaration nodes) are represented using the following constructor symbols:

constructor `xml::`**`required`**
: a required attribute, i.e., the user must specify this

constructor `xml::`**`implied`**
: an implied attribute, i.e., the user does not have to specify this

constructor `xml::`**`default`** val
: an attribute with the given default value `val`

constructor `xml::`**`fixed`** val
: an attribute with the given fixed value `val`

23.5 Operations

This module provides all operations necessary to create, inspect and manipulate XML documents residing either in memory or on disk. Operations for formatting XML documents using XSLT stylesheets are also available.

23.5.1 Document Operations

The following functions allow you to create new XML documents, load them from or save them to a file or a string, and provide general information about a document.

`xml::`**`new_doc`** version dtd info

This function creates an XML document. It returns a pointer to the new document. `version` is a string denoting the XML version (or `""` to indicate the default). `info` is the node info of the root node (which should denote an element node). `dtd` denotes the document type which can be `()` to denote an empty DTD, a string (the URI/filename of the DTD), or a pair `(pubid,sysid)` where `pubid` denotes the public identifier of the DTD and `sysid` its system identifier (URI).

Note that only simple kinds of documents with an internal DTD can be created this way. Use the `load_file` or `load_string` function below to create a skeleton document if a more elaborate prolog is required.

`xml::`**`load_file`** name flags

`xml::`**`load_string`** s flags

Load an XML document from a file `name` or a string `s`. `flags` denotes the parser flags, a bitwise disjunction of any of the following constants, or 0 for the default:

- `xml::DTDLOAD`: load DTD
- `xml::DTDVALID`: validate
- `xml::PEDANTIC`: pedantic parse
- `xml::SUBENT`: substitute entities
- `xml::NOBLANKS`: suppress blank nodes

The return value is the document pointer. These operations may also fail if there is a fatal error parsing the document.

`xml::`**`save_file`** name doc encoding compression

`xml::`**`save_string`** doc

Save an XML document `doc` to a file or a string. When saving to a file, `encoding` denotes the desired encoding (or `""` for the default), `compression` the desired level of zlib compression (0 means none, 9 is maximum, -1 indicates the default). Note that with `xml::save_string`, the result is always encoded as UTF-8.

`xml::`**`doc_info`** doc

Retrieve general information about a document. Returns a tuple `(version,encoding,url,compression,standalone)`, where `version` is the XML version of the document, `encoding` the external encoding (if any), `url` the name/location of the document (if any), `compression` the level of zlib compression, and `standalone` is a flag indicating whether the document contains any external markup declarations "which affect the information passed from the XML processor to the application", see the section on the standalone document declaration in the XML spec for details. (Apparently, in libxml2 `standalone` is either a truth value or one of the special values -1, indicating that there's no XML declaration in the prolog, or -2, indicating that there's an XML declaration but no `standalone` attribute.)

`xml::`**`int_subset`** doc

`xml::`**`ext_subset`** doc

Retrieve the internal and external DTD subset of a document. Returns a `doctype` node

(fails if there's no corresponding DTD).

Example

Read the sample.xml document distributed with the sources (ignoring blank nodes) and retrieve the document info:

```
> using xml;
> let sample = xml::load_file "sample.xml" xml::NOBLANKS;
> xml::doc_info sample;
"1.0","","sample.xml",0,-2
```

23.5.2 Traversing Documents

These operations are used to traverse the document tree, i.e., the nodes of the document. They take either a document pointer `doc` or a node pointer `node` as argument, and yield a corresponding node pointer (or a document pointer, in the case of `xml::doc`). The node pointers can then be used with the Node Information and Node Manipulation operations described below.

`xml::`**`root`** doc
: the root node of `doc`

`xml::`**`doc`** node
: the document `node` belongs to

`xml::`**`parent`** node
: the parent of `node`

`xml::`**`first`** node
`xml::`**`last`** node
: first and last child node

`xml::`**`next`** node
`xml::`**`prev`** node
: next and previous sibling

`xml::`**`first_attr`** node
`xml::`**`last_attr`** node
: first and last attribute node

All these operations fail if the corresponding target node does not exist, or if the type of the given node is not supported by this implementation.

There are also two convenience functions to retrieve the children and attribute nodes of a node:

`xml::`**`children`** node
: returns the list of all child nodes of `node`

`xml::`**`attrs`** node
: returns the list of all attribute nodes of `node`

Moreover, given a node pointer `node`, `node!i` can be used to retrieve the `i`th child of `node`.

Example

Peek at the root node of the sample document and its children:

```
> let r = xml::root sample; r;
#<pointer 0xe15e10>
> xml::node_info r;
xml::element "story" [] []
> #xml::children r;
5
> xml::node_info (r!0);
xml::cdata "<greeting>Hello, world!</greeting>"
```

23.5.3 Node Information

These operations retrieve information about the nodes of an XML document.

`xml::`**`select`** doc xpath
`xml::`**`select`** doc (xpath,ns)
: Retrieve nodes using an XPath specification. Given an XPath (a string) `xpath`, this operation returns the list of all matching nodes in the given document `doc`. You can also specify a node as the first argument, in which case the document of the given node is searched and paths are interpreted relative to the given node (rather than the root node of the document).

  Moreover, instead of just an XPath you can also specify a pair `(xpath,ns)` consisting of an XPath `xpath` and a list `ns` of `prefix=>uri` string pairs which describe the namespaces to be recognized in the XPath expression. This is necessary to select nodes by qualified tag or attribute names. Note that only the namespace URIs must match up with those used in the queried document; the corresponding namespace prefixes can be chosen freely, so you can use whatever prefixes are convenient to formulate the XPath query. However, for each namespace prefix used in the XPath expression (not the document!), there *must* be a corresponding binding in the `ns` list. Otherwise the underlying libxml2 function will complain about an undefined namespace prefix and `xml::select` will fail.

`xml::`**`node_info`** node
: Retrieve the node data from `node`. Returns a node info value, as described in Data Structure above. Fails if the node does not belong to one of the supported node types.

`xml::`**`is_blank_node`**
: Checks whether a node is a blank node (empty or whitespace only) and thus possibly ignorable.

`xml::`**`node_base`** node
: Returns the base URI of the given node.

`xml::`**`node_path`** node
: Returns the path of a node in the document, in the format accepted by `select`.

`xml::node_content` node
: Returns the text carried by the node, if any (after entity substitution).

In addition, you can retrieve and change attributes of element nodes with the following operations:

`xml::node_attr` node name
: Retrieves the value of the attribute with the given `name` (after entity substitution).

`xml::set_node_attr` node name value
`xml::unset_node_attr` node name
: Sets or unsets an attribute value.

Examples

Set and unset a node attribute:

```
> xml::set_node_attr r "foo" "4711";
()
> xml::node_info r;
xml::element "story" [] ["foo"=>"4711"]
> xml::node_attr r "foo";
"4711"
> xml::unset_node_attr r "foo";
()
> xml::node_info r;
xml::element "story" [] []
```

The `select` function is *very* powerful, and probably the single most important operation of this module if you want to extract information from an existing XML document without traversing the entire structure. Here is a very simple example of its use:

```
> [xml::node_content n, xml::node_path n | n = xml::select sample "//author"];
[("John Fleck","/story/storyinfo/author")]
```

Note that if the XPath expression contains qualified names, the corresponding namespace prefixes and their URIs must be given in the second argument along with the XPath, as follows:

```
xml::select doc ("//foo:bar", ["foo"=>"http://www.foo.org"]);
```

23.5.4 Node Manipulation

These operations enable you to manipulate the document structure by adding a new node to the document tree (specified through its node info), and by removing (unlinking) existing nodes from the tree.

`xml::replace` node info
: Add the new node specified by `info` in place of the given node `node`.

`xml::add_first` node info

`xml::`**`add_last`** node info
: Add the new node as the first or last child of `node`, respectively.

`xml::`**`add_next`** node info
`xml::`**`add_prev`** node info
: Add the new node as the next or previous sibling of `node`, respectively.

The operations above all return a pointer to the new XML node object.

`xml::`**`unlink`** node
: Deletes an existing node from the document tree. Returns `()`.

Examples

Replace the first child of the root node in the sample document:

```
> xml::node_info (r!0);
xml::cdata "<greeting>Hello, world!</greeting>"
> xml::replace (r!0) (xml::text "bla bla");
#<pointer 0xd40d80>
> xml::node_info (r!0);
xml::text "bla bla"
```

Delete that node:

```
> xml::unlink (r!0);
()
> xml::node_info (r!0);
xml::comment "This is a sample document for testing the xml interface."
```

23.5.5 Transformations

The following operations provide basic XSLT support. As already mentioned, stylesheets are represented as pointers to the xsltStylesheet structure provided by libxslt. Note that, in difference to XML document pointers, this is an opaque type, i.e., there is no direct means to inspect and manipulate parsed stylesheets in memory using the operations of this module. However, a stylesheet is just a special kind of XML document and thus can be manipulated after reading the stylesheet as an ordinary XML document. The `load_stylesheet` function then allows you to convert the document pointer to an XSLT Stylesheet object.

Applying a stylesheet to an XML document generally involves the following steps:

1. Load and parse the stylesheet using `load_stylesheet`. The parameter to `load_stylesheet` can be either the name of a stylesheet file or a corresponding document pointer. The function returns a pointer to the stylesheet object which is used in the subsequent processing.
2. Invoke `apply_stylesheet` on the stylesheet and the target document. This returns a new document containing the transformed XML document.
3. Run `save_result_file` or `save_result_string` on the result and the stylesheet to save the transformed document in a file or a string.

Here is a brief summary of the XSLT operations. Please check the XSLT documentation for details of the transformation process.

`xslt::`**`load_stylesheet`** x
: Load a stylesheet. `x` can be either an XML document pointer, or a string denoting the desired `.xsl` file.

`xslt::`**`apply_stylesheet`** style doc params
: Apply the stylesheet `style` to the given document `doc` with the given parameters `params`. The third argument is a (possibly empty) list of `key=>value` string pairs which allows you to pass additional parameters to the stylesheet.

`xslt::`**`save_result_file`** name doc style compression
`xslt::`**`save_result_string`** doc style
: Save the transformation result `doc` obtained with the stylesheet `style` to a file or a string. This works pretty much like `save_file` or `save_string`, but also keeps track of some output-related information contained in the stylesheet.

Example

Load the recipes.xml document and the recipes.xsl stylesheet distributed with the sources:

```
> let recipes = xml::load_file "recipes.xml" xml::DTDVALID;
> let style = xslt::load_stylesheet "recipes.xsl";
```

Apply the stylesheet to the document and save the result in a html file:

```
> let res = xslt::apply_stylesheet style recipes [];
> xslt::save_result_file "recipes.html" res style 0;
()
```

That's all. You can now have a look at recipes.html in your favourite web browser.

Chapter 24

pure-g2

Version 0.2, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This is a straight wrapper of the g2 graphics library, see http://g2.sf.net/.

License: BSD-style, see the COPYING file for details.

Get the latest source from http://pure-lang.googlecode.com/files/pure-g2-0.2.tar.gz.

g2 is a simple, no-frills 2D graphics library, distributed under the LGPL. It's easy to use, portable and supports PostScript, X11, PNG and Win32. Just the kind of thing that you need if you want to quickly knock out some basic graphics, and whipping out the almighty OpenGL or GTK/Cairo seems overkill.

To use this module, you need to have libg2 installed as a shared library (libg2.so, .dll etc.) in a place where the Pure interpreter can find it.

Documentation still needs to be written, so for the time being please see g2.pure and have a look at the examples provided in the distribution.

Run `make install` to copy g2.pure to the Pure library directory. This tries to guess the prefix under which Pure is installed; if this doesn't work, you'll have to set the prefix variable in the Makefile accordingly.

The Makefile also provides the following targets:

- `make examples` compiles the examples to native executables.
- `make clean` deletes the native executables for the examples, as well as some graphics files which are produced by running g2_test.pure.
- `make generate` regenerates the g2.pure module. This requires that you have pure-gen installed, as well as the g2 header files (you can point pure-gen to the prefix under which g2 is installed with the g2prefix variable in the Makefile). This step shouldn't normally be necessary, unless you find that the provided wrapper doesn't work with your g2 version. The g2.pure in this release has been generated from g2 0.72.

Chapter 25

Pure OpenGL Bindings

Version 0.8, June 26, 2012

Scott Dillard
Albert Graef <Dr.Graef@t-online.de>

These are fairly complete Pure bindings for the OpenGL graphics library, which allow you to do 2D and 3D graphics programming with Pure. The bindings should work out of the box on most contemporary systems which have OpenGL drivers installed, thanks to Scott's on-demand loading code for the GL functions, which accounts for the fact that different GL implementations will export different functions. (Mostly to account for Microsoft's Museum of Ancient OpenGL History, otherwise known as opengl32.dll.)

Information about OpenGL can be found at: http://www.opengl.org/

As of pure-gl 0.5, the bindings are now generated using pure-gen instead of Scott's original OpenGL-specific generator. The stuff needed to do this is included (except pure-gen, which is a separate package available from the Pure website), so that you can regenerate the bindings if necessary.

25.1 Copying

Copyright (c) 2009, Scott E Dillard
Copyright (c) 2009, Albert Graef
Copyright (c) 2002-2005, Sven Panne

pure-gl is distributed under a 3-clause BSD-style license, please see the accompanying COPYING file for details.

25.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-gl-0.8.tar.gz.

Normally you just run `make && sudo make install`, as with the other Pure modules. (See The Makefile for further options.) This doesn't regenerate the bindings and so can be done on any system which has Pure, OpenGL and a C compiler installed.

If you miss some vendor-specific OpenGL functionality which is in your system's header files but not in the distributed bindings, with some effort you can fix that yourself by regenerating the bindings, see below.

25.3 Using the GL Bindings

The bindings mainly consist of 3 Pure files: GL.pure, GLU.pure and GLUT.pure.

In your Pure program, write something like:

```
using GL, GLU, GLUT;
```

GL.pure covers OpenGL up through version 2.1. To get access to extensions, you include GL_XXX.pure where XXX is the extensions suffix. Currently, there are GL_ARB.pure, GL_EXT.pure, GL_NV.pure and GL_ATI.pure, which should cover about 99% of the useful extensions out there. If you need more than that, it is straightforward to tweak the Makefile to scrape some of the more esoteric extensions from your headers. All OpenGL functions are loaded on first use. If your OpenGL implementation does not define a given function, a `gl_unsupported` exception is thrown with the name of the function as its only argument.

The functions are in namespaces GL, GLU and GLUT respectively. Functions are in curried form, i.e.:

```
GL::Vertex3d 1.0 2.0 3.0;
```

GL enumerants are in uppercase, as in C:

```
GL::Begin GL::LINE_STRIP;
```

Currently, if the GLU or GLUT bindings reference a function that your DLL does not contain, it echoes this to stdout. I'm working on a way to supress this.

Some examples can be found in the examples subdirectory. This also includes a wrapper of Rasterman's imlib2 library (also generated with pure-gen), and an example which uses this to render an image as a texture.

The examples/flexi-line directory contains Eduardo Cavazos' port of the flexi-line demo. Run `pure flexi-line-auto.pure`, sit back and enjoy. There's also an interactive version of the demo available in flexi-line.pure.

25.4 Regenerating the Bindings

You need to have pure-gen installed to do this.

Also make sure that you have the OpenGL headers installed. By default, the Makefile assumes that they are in the GL subdirectory of /usr/include, you can set the `glpath` variable in the Makefile accordingly to change this. (Set `glpath` to the path under which the GL subdirectory resides, not to the GL subdirectory itself. See below for an example.) Note that on Linux systems, /usr/include/GL usually contains the MESA headers. If available, you may want to use your GPU vendor's headers instead, to get all the extensions available on your system.

Alternatively, you can also just put the headers (gl.h, glext.h, glu.h, glut.h, and any other OpenGL headers that get #included in those) into the GL subdirectory of the pure-gl sources, by copying them over or creating symbolic links to them. This is particularly useful for maintainers, who may want to use a "staged" header set which is different from the installed OpenGL headers. The "." directory will always be searched first, so you can also just put the vendor-specific headers there. For instance, if you're like Scott and you use Ubuntu with an Nvidia GPU, then you can do this:

```
cd pure-gl/GL
ln -s /usr/share/doc/nvidia-glx-new-dev/include/GL/gl.h
ln -s /usr/share/doc/nvidia-glx-new-dev/include/GL/glext.h
```

Finally, the Makefile also assumes that you have freeglut (an improved GLUT replacement) installed and want all the extensions offered by freeglut. To use the vanilla GLUT without the extensions instead, you only have to change the value of the `source` variable in the Makefile from GL/all_gl_freeglut.h to GL/all_gl.h. If you use openglut instead of freeglut you will have to change the GL/all_gl_freeglut.h file accordingly.

Once you have set up things to your liking, you can regenerate the bindings by running make as follows:

```
make generate
```

If you need a custom path to the OpenGL headers as described above (say, /usr/local/include) then do this instead:

```
make generate glpath=/usr/local/include
```

If you're lucky, this will regenerate all the GL*.pure and GL*.c files, and recompile the shared module from the GL*.c files after that. This shared module, instead of the OpenGL libraries themselves, is what gets loaded by the Pure modules.

If you're not so lucky, save a complete build log with all the error messages and ask on the pure-lang mailing list for help.

See the "Generator stuff" section in the Makefile for further options. Adding a rule for other extensions should be easy, just have a look at an existing one (e.g., GL_EXT.c) and modify it accordingly.

Chapter 26

Pure GTK+ Bindings

Version 0.11, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

pure-gtk is a collection of bindings to use the GTK+ GUI toolkit version 2.x with Pure, see http://www.gtk.org. The bindings include the gtk (+gdk), glib, atk, cairo and pango libraries, each in their own Pure module.

At present these are just straight 1-1 wrappers of the C libraries, created with pure-gen. So they still lack some convenience, but they are perfectly usable already, and a higher-level API for accessing all the functionality will hopefully become available in time. In fact *you* can help make that happen. :) So please let me know if you'd like to give a helping hand in improving pure-gtk.

26.1 Copying

Copyright (c) 2008-2011 by Albert Graef.

pure-gtk is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

26.2 Installation

You can get the latest source from http://pure-lang.googlecode.com/files/pure-gtk-0.11.tar.gz.

For Windows users, a ready-made package in msi format is available from http://pure-lang.googlecode.com/files/pure-gtk-0.11.msi.

To install from source, do the usual `make && sudo make install` (see the Makefile for further options). This needs Pure and the GTK header files and libraries. You'll also need *pure-ffi* for running the examples.

NOTE: The source release was prepared with GTK+ 2.24.4 on Ubuntu 11.04. If you're seeing a lot of warnings and/or errors when compiling or loading the modules, your GTK headers are probably much different from these. In that case you should run `make generate` to regenerate the bindings; for this you also need to have pure-gen installed. (If you already have pure-gen then it's a good idea to do this anyway.)

26.3 Usage

See examples/hello.pure for a basic example. The files uiexample.pure and uiexample.glade show how to run a GUI created with the Glade-3 interface builder. This needs a recent version of the GtkBuilder API to work. (If you're still running Glade-2 and an older GTK+ version, you might want to use the older libglade interface instead. Support for that is in the Makefile, but it's not enabled by default.) NOTE: The examples start up much faster when they are compiled to native executables. To do this, just run `make examples` after `make`. (Be patient, this takes a while.)

pure-gtk can be discussed on the Pure mailing list at: http://groups.google.com/group/pure-lang

Chapter 27

pure-tk

Version 0.3, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

Pure's Tcl/Tk interface.

27.1 Introduction

This module provides a basic interface between Pure and Tcl/Tk. The operations of this module allow you to execute arbitrary commands in the Tcl interpreter, set and retrieve variable values in the interpreter, and invoke Pure callbacks from Tcl/Tk.

A recent version of Tcl/Tk is required (8.0 or later should do). You can get this from http://www.tcl.tk. Both releases in source form and binary releases for Windows and various Unix systems are provided there.

Some information on how to use this module can be found below. But you'll find that pure-tk is very easy to use, so you might just want to look at the programs in the examples folder to pick it up at a glance. A very basic example can be found in tk_hello.pure; a slightly more advanced example of a tiny but complete Tk application is in tk_examp.pure.

pure-tk also offers special support for Peter G. Baum's Gnocl extension which turns Tcl into a frontend for GTK+ and Gnome. If you have Gnocl installed then you can easily create GTK+/Gnome applications, either from Tcl sources or from Glade UI files, using the provided gnocl.pure module. See the included uiexample.pure and the accompanying Glade UI file for a simple example. Also, some basic information on using Gnocl with pure-tk can be found in the Tips and Tricks section below.

One nice thing about Tcl/Tk is that it provides a bridge to a lot of other useful libraries. A prominent example is VTK, a powerful open-source 3D visualization toolkit which comes with full Tcl/Tk bindings. The examples directory contains a simple example (earth.pure

and earth.tcl) which shows how you can employ these bindings to write cool animated 3D applications using either Tk or Gnocl as the GUI toolkit.

27.2 Copying

Copyright (c) 2010 by Albert Gräf, all rights reserved. pure-tk is distributed under a BSD-style license, see the COPYING file for details.

27.3 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-tk-0.3.tar.gz.

As with the other addon modules for Pure, running `make && sudo make install` should usually do the trick. This requires that you have Pure and Tcl/Tk installed. `make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix. Please see the Makefile for details.

Note: When starting a new interpreter, the Tcl/Tk initialization code looks for some initialization files which it executes before anything else happens. Usually these files will be found without any further ado, but if that does not happen automatically, you must set the TCL_LIBRARY and TK_LIBRARY environment variables to point to the Tcl and Tk library directories on your system.

All programs in the examples subdirectory have been set up so that they can be compiled to native executables, and a Makefile is provided in that directory to handle this. So after installing pure-tk you just need to type `make` there to compile the examples. (This step isn't necessary, though, you can also just run the examples with the Pure interpreter as usual.)

27.4 Basic Usage

`tk` cmd
: execute a Tcl command

You can submit a command to the Tcl interpreter with `tk cmd` where `cmd` is a string containing the command to be executed. If the Tcl command returns a value (i.e., a nonempty string) then `tk` returns that string, otherwise it returns `()`.

`tk` also starts a new instance of the Tcl interpreter if it is not already running. To stop the Tcl interpreter, you can use the `tk_quit` function.

`tk_quit`
: stop the Tcl interpreter

Note that, as far as pure-tk is concerned, there's only one Tcl interpreter per process, but of course you can create secondary interpreter instances in the Tcl interpreter using the appropriate Tcl commands.

Simple dialogs can be created directly using Tk's `tk_messageBox` and `tk_dialog` functions. For instance:

```
tk "tk_dialog .warning \"Warning\" \"Are you sure?\" warning 0 Yes No Cancel";
```

Other kinds of common dialogs are available; see the Tcl/Tk manual for information.

For more elaborate applications you probably have to explicitly create some widgets, add the appropriate callbacks and provide a main loop which takes care of processing events in the Tcl/Tk GUI. We discuss this in the following.

27.5 Callbacks

pure-tk installs a special Tcl command named `pure` in the interpreter which can be used to implement callbacks in Pure. This command is invoked from Tcl as follows:

```
pure function args ...
```

It calls the Pure function named by the first argument, passing any remaining (string) arguments to the callback. If the Pure callback returns a (nonempty) string, that value becomes the return value of the `pure` command, otherwise the result returned to the Tcl interpreter is empty.

Pure callbacks are installed on Tk widgets just like any other, just using the `pure` command as the actual callback command. For instance, you can define a callback which gets invoked when a button is pushed as follows:

```
using tk, system;
tk "button .b -text {Hello, world!} -command {pure hello}; pack .b";
hello = puts "Hello, world!";
```

27.6 The Main Loop

tk_main
: call the Tk main loop

The easiest way to provide a main loop for your application is to just call `tk_main` which keeps processing events in the Tcl interpreter until the interpreter is exited. You can terminate the interpreter in a Pure callback by calling `tk_quit`. Thus a minimalistic Tcl/Tk application coded in Pure may look as follows:

```
using tk;
tk "button .b -text {Hello, world!} -command {pure tk_quit}; pack .b";
tk_main;
```

The main loop terminates as soon as the Tcl interpreter is exited, which can happen, e.g., in response to a callback which invokes the `tk_quit` function (as shown above) or Tcl code which destroys the main window (`destroy .`). The user can also close the main window from the window manager in order to exit the main loop.

27.7 Accessing Tcl Variables

tk_set var val
tk_unset var
tk_get var
    set and get Tcl variables

pure-tk allows your script to set and retrieve variable values in the Tcl interpreter with the `tk_set`, `tk_unset` and `tk_get` functions. This is useful, e.g., to change the variables associated with entry and button widgets, and to retrieve the current values from the application. For instance:

```
> tk_set "entry_val" "some string";
"some string"
> tk_get "entry_val";
"some string"
> tk_unset "entry_val";
()
> tk_get "entry_val";
tk_get "entry_val"
```

Note that `tk_set` returns the assigned value, so it is possible to chain such calls if several variables have to be set to the same value:

```
> tk_set "foo" $ tk_set "bar" "yes";
"yes"
> map tk_get ["foo","bar"];
["yes","yes"]
```

27.8 Conversions Between Pure and Tcl Values

As far as pure-tk is concerned, all Tcl values are strings (in fact, that's just what they are at the Tcl language level, although the Tcl interpreter uses more elaborate representations of objects such as lists internally). There are no automatic conversions of any kind. Thus, the arguments passed to a Pure callback and the result returned by `tk` are simply strings in Pure land. The same holds for the `tk_set` and `tk_get` functions.

However, there are a few helper functions which can be used to convert between Tcl and Pure data. First, the following operations convert Pure lists to corresponding Tcl lists and vice versa:

tk_join xs

tk_split s
: convert between Pure and Tcl lists

```
> tk_join ["0","1.0","Hello, world!"];
"0 1.0 {Hello, world!}"
> tk_split ans;
["0","1.0","Hello, world!"]
```

The tk_str and tk_val operations work in a similar fashion, but they also do automatic conversions for numeric values (ints, bigints and doubles):

tk_str xs
tk_val s
: convert between Pure and Tcl values with numeric conversions

```
> tk_str [0,1.0,"Hello, world!"];
"0 1.0 {Hello, world!}"
> tk_val ans;
[0,1.0,"Hello, world!"]
```

In addition, these operations also convert single atomic values:

```
> tk_str 1.0;
"1.0"
> tk_val ans;
1.0
```

27.9 Tips and Tricks

Here are a few other things that are worth keeping in mind when working with pure-tk.

- Errors in Tcl/Tk commands can be handled by giving an appropriate definition of the `tk_error` function, which is invoked with an error message as its single argument. For instance, the following implementation of `tk_error` throws an exception:

  ```
  tk_error msg = throw msg;
  ```

  If no definition for this function is provided, then errors cause a literal `tk_error msg` expression to be returned as the result of the `tk` function. You can then check for such results and take an appropriate action.

- The Tcl interpreter, when started, displays a default main window, which is required by most Tk applications. If this is not desired (e.g., if only the basic Tcl commands are needed), you can hide this window using a `tk "wm withdraw ."` command. To redisplay the window when it is needed, use the `tk "wm deiconify ."` command. It is also common practice to use `wm withdraw` and `wm deiconify` while creating the widgets of an application, in order to reduce "flickering".

- Instead of calling `tk_main`, you can also code your own main loop in Pure as follows:

```
main = do_something $$ main if tk_ready;
     = () otherwise;
```

Note that the `tk_ready` function checks whether the Tcl interpreter is still up and running, after processing any pending events in the interpreter. This setup allows you to do your own custom idle processing in Pure while the application is running. However, you have to be careful that your `do_something` routine runs neither too short nor too long (a few milliseconds should usually be ok). Otherwise your main loop may turn into a busy loop and/or the GUI may become very sluggish and unresponsive. Thus it's usually better to just call `tk_main` and do any necessary background processing using the Tcl interpreter's own facilities (e.g., by setting up a Pure callback with the Tcl `after` command).

- The `tk` function can become rather tedious when coding larger Tk applications. Usually, you will prefer to put the commands making up your application into a separate Tcl script. One way to incorporate the Tcl script into your your Pure program is to use the Tcl `source` command, e.g.:

  ```
  tk "source myapp.tcl";
  ```

  However, this always requires the script to be available at runtime. Another method is to read the script into a string which is assigned to a Pure constant, and then invoke the `tk` command on this string value:

  ```
  using system;
  const ui = fget $ fopen "myapp.tcl" "r";
  tk ui;
  ```

  This still reads the script at runtime if the Pure program is executed using the Pure interpreter. However, you can now compile the Pure program to a native executable (see the Pure manual for details on this), in which case the text of the Tcl script is included verbatim in the executable. The compiled program can then be run without having the original Tcl script file available:

  ```
  $ pure -c myapp.pure -o myapp
  $ ./myapp
  ```

  This is also the method to use for running existing Tk applications, e.g., if you create the interface using some interface builder like vtcl.

- The Tcl `package` command allows you to load additional extensions into the Tcl interpreter at runtime. For instance:

  ```
  tk "package require Gnocl";
  ```

  This loads Peter G. Baum's Gnocl extension which turns Tcl into a frontend for GTK+ and Gnome. In fact, pure-tk includes a special module to handle the nitty-gritty details of creating a GTK+/Gnome application from a Glade UI file and set up Pure callbacks as specified in the UI file. To use this, just import the gnocl.pure module into your Pure scripts:

```
using gnocl;
```

Note that the Glade interface requires that you have a fairly recent version of Gnocl installed (Gnocl 0.9.94g has been tested). The other facilities provided by the gnocl.pure module should also work with older Gnocl versions such as Gnocl 0.9.91. Please see the gnocl.pure module and the corresponding examples included in the sources for more information.

- The Tcl `exit` procedure, just as in tclsh or wish, causes exit from the current process. Since the Tcl interpreter hosted by the pure-tk module runs as part of a Pure program and not as a separate child process, this might not be what you want. If you'd like `exit` to only exit the Tcl interpreter, without exiting the Pure program, you can redefine the `exit` procedure, e.g., as follows:

  ```
  tk "proc exit { {returnCode 0} } { pure tk_quit }";
  ```

  If you want to do something with the exit code provided by `exit`, you will have to provide an appropriate callback function, e.g.:

  ```
  tk "proc exit { {returnCode 0} } { pure quit_cb $returnCode }";
  ```

  A suitable implementation of `quit_cb` might look as follows:

  ```
  quit_cb 0 = puts "Application exited normally." $$ tk_quit;
  quit_cb n = printf "Application exited with exit code %d.\n" n $$
              tk_quit otherwise;
  ```

- If you need dialogs beyond the standard kinds of message boxes and common dialogs, you will have to do these yourself using a secondary toplevel. The dialog toplevel is just like the main window but will only be shown when the application needs it. You can construct both non-modal and modal dialogs this way, the latter can be implemented using Tk's `grab` command.

Chapter 28

faust2pd: Pd Patch Generator for Faust

Version 2.5, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This package contains software which makes it easier to use Faust DSPs with Pd and the Pure programming language. The main component is the faust2pd script which creates GUI wrappers for Faust DSPs. The package also includes a bunch of examples. The software is distributed under the GPL; see the COPYING file for details.

Note: This faust2pd version is written in Pure and was ported from an earlier version written in Pure's predecessor Q. The version of the script provided here should be 100% backward-compatible to those previous versions, except for the following changes:

- The (rarely used) -f (a.k.a. –fake-buttons) option was renamed to -b.
- A new -f (a.k.a. –font-size) option was added to change the font size of the GUI elements.
- Most command line options can now also be specified using special `[pd:...]` tags in the Faust source.

Also note that you can now run the script on 64 bit systems (Q never worked there).

As of version 2.1, the faust2pd script is now compiled to a native executable before installation. This makes the program start up much faster, which is a big advantage because most xml files don't take long to be processed.

28.1 Copying

Copyright (c) 2009-2011 by Albert Graef.

faust2pd is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

faust2pd is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

28.2 Requirements

faust2pd is known to work on Linux, Mac OS X, and Windows, and there shouldn't be any major roadblocks preventing it to work on other systems supported by Pure.

The faust2pd script is written in the Pure programming language and requires Pure's XML module, so you need to install these to make it work. Install the latest pure*.tar.gz and pure-xml*.tar.gz packages and you should be set. (Pure 0.47 or later is required.) Also make sure that the LLVM base package is installed, as described in the Pure INSTALL file, some LLVM utilities are needed to make Pure's batch compiler work.

To run the seqdemo example, you'll also need the Pd Pure external (pd-pure*.tar.gz), also available at the Pure website.

To compile the examples, you'll need GNU C++ and make, Pd and, of course, Faust. Make sure you get a recent version of Faust; Faust releases >0.9.8 include the puredata architecture necessary to create Pd externals from Faust DSPs.

28.3 Installation

Get the latest source from http://pure-lang.googlecode.com/files/faust2pd-2.5.tar.gz.

Run `make` and `make install` to compile and install the faust2pd program on your system. You can set the installation prefix by running make as `make install prefix=/some/path`. Default installation prefix is /usr/local, faust2pd is installed in the bin directory below that.

Optionally, you can also run `make install-pd` to copy the supporting Pd abstractions (faust-*.pd) to your lib/pd/extra directory, so that you can use the patches generated by faust2pd without copying these abstractions to your working directory. The Makefile tries to guess the prefix of your Pd installation, if it guesses wrong, you can specify the prefix explicitly by running make as `make install-pd pdprefix=/some/path`.

The included faustxml.pure script provides access to Faust-generated dsp descriptions in xml files to Pure scripts. This module is described in its own appendix below. It may have uses beyond faust2pd, but isn't normally installed. If you want to use this module, you can just copy it to your Pure library directory.

28.4 Quickstart

Run `make examples` to compile the Faust examples included in this package to corresponding Pd plugins. After that you should be able to run the patches in the various subdirectories of the examples directory. Everything is set up so that you can try the examples "in-place", without installing anything except the required software as noted in Requirements above. You can also run `make realclean` before `make` to regenerate everything from scratch (this requires faust2pd, so this will only work if you already installed the Pure interpreter).

Faust Pd plugins work in much the same way as the well-known plugin~ object (which interfaces to LADSPA plugins), except that each Faust DSP is compiled to its own Pd external. Under Linux, the basic compilation process is as follows (taking the freeverb module from the Faust distribution as an example):

```
# compile the Faust source to a C++ module using the "puredata" architecture
faust -a puredata.cpp freeverb.dsp -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp -o freeverb~.pd_linux
```

By these means, a Faust DSP named `xyz` with n audio inputs and m audio outputs becomes a Pd object `xyz~` with n+1 inlets and m+1 outlets. The leftmost inlet/outlet pair is for control messages only. This allows you to inspect and change the controls the unit provides, as detailed below. The remaining inlets and outlets are the audio inputs and outputs of the unit, respectively. For instance, `freeverb.dsp` becomes the Pd object `freeverb~` which, in addition to the control inlet/outlet pair, has 2 audio inputs and outputs.

When creating a Faust object it is also possible to specify, as optional creation parameters, an extra unit name (this is explained in the following section) and a sample rate. If no sample rate is specified explicitly, it defaults to the sample rate at which Pd is executing. (Usually it is not necessary or even desirable to override the default choice, but this might occasionally be useful for debugging purposes.)

As already mentioned, the main ingredient of this package is a Pure script named "faust2pd" which allows you to create Pd abstractions as "wrappers" around Faust units. The wrappers generated by faust2pd can be used in Pd patches just like any other Pd objects. They are much easier to operate than the "naked" Faust plugins themselves, as they also provide "graph-on-parent" GUI elements to inspect and change the control values.

The process to compile a plugin and build a wrapper patch is very similar to what we've seen above. You only have to add the -xml option when invoking the Faust compiler and run faust2pd on the resulting XML file:

```
# compile the Faust source and generate the xml file
faust -a puredata.cpp -xml freeverb.dsp -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp -o freeverb~.pd_linux
# generate the Pd patch from the xml file
faust2pd freeverb.dsp.xml
```

Please see Wrapping DSPs with faust2pd below for further details.

Note that, just as with other Pd externals and abstractions, the compiled .pd_linux modules and wrapper patches must be put somewhere where Pd can find them. To these ends you can either move the files into the directory with the patches that use the plugin, or you can put them into the lib/pd/extra directory or some other directory on Pd's library path for system-wide use.

Also, faust2pd-generated wrappers use a number of supporting abstractions (the faust-*.pd files in the faust2pd sources), so you have to put these into the directory of the main patch or install them under lib/pd/extra as well. (The `make pd-install` step does the latter, see Installation above.)

28.5 Control Interface

The control inlet of a Faust plugin understands messages in one of the following forms:

- `bang`, which reports all available controls of the unit on the control outlet, in the form: `type name val init min max step`, where `type` is the type of the control as specified in the Faust source (`checkbox`, `nentry`, etc.), `name` its (fully qualified) name, `val` the current value, and `init`, `min`, `max`, `step` the initial value, minimum, maximum and stepsize of the control, respectively.
- `foo 0.99`, which sets the control `foo` to the value 0.99, and outputs nothing.
- Just `foo`, which outputs the (fully qualified) name and current value of the `foo` control on the control outlet.

Control names can be specified in their fully qualified form, like e.g. `/gnu/bar/foo` which indicates the control `foo` in the subgroup `bar` of the topmost group `gnu`, following the hierarchical group layout defined in the Faust source. This lets you distinguish between different controls with the same name which are located in different groups. To find out about all the controls of a unit and their fully qualified names, you can bang the control inlet of the unit as described above, and connect its control outlet to a `print` object, which will cause the descriptions of all controls to be printed in Pd's main window. (The same information can also be used, e.g., to initialize GUI elements with the proper values. Patches generated with faust2pd rely on this.)

You can also specify just a part of the control path (like `bar/foo` or just `foo` in the example above) which means that the message applies to *all* controls which have the given pathname as the final portion of their fully qualified name. Thus, if there is more than one `foo` control in different groups of the Faust unit then sending the message `foo` to the control inlet will report the fully qualified name and value for each of them. Likewise, sending `foo 0.99` will set the value of all controls named `foo` at once.

Concerning the naming of Faust controls in Pd you should also note the following:

- A unit name can be specified at object creation time, in which case the given symbol is used as a prefix for all control names of the unit. E.g., the control `/gnu/bar/foo` of an object `baz~` created with `baz~ baz1` has the fully qualified name `/baz1/gnu/bar/foo`.

This lets you distinguish different instances of an object such as, e.g., different voices of a polyphonic synth unit.

- Pd's input syntax for symbols is rather restrictive. Therefore group and control names in the Faust source are mangled into a form which only contains alphanumeric characters and hyphens, so that the control names are always legal Pd symbols. For instance, a Faust control name like `meter #1 (dB)` will become `meter-1-dB` which can be input directly as a symbol in Pd without any problems.
- "Anonymous" groups and controls (groups and controls which have empty labels in the Faust source) are omitted from the path specification. E.g., if `foo` is a control located in a main group with an empty name then the fully qualified name of the control is just `/foo` rather than `//foo`. Likewise, an anonymous control in the group `/foo/bar` is named just `/foo/bar` instead of `/foo/bar/`.

Last but not least, there is also a special control named `active` which is generated automatically for your convenience. The default behaviour of this control is as follows:

- When `active` is nonzero (the default), the unit works as usual.
- When `active` is zero, and the unit's number of audio inputs and outputs match, then the audio input is simply passed through.
- When `active` is zero, but the unit's number of audio inputs and outputs do *not* match, then the unit generates silence.

The `active` control frequently alleviates the need for special "bypass" or "mute" controls in the Faust source. However, if the default behaviour of the generated control is not appropriate you can also define your own custom version of `active` explicitly in the Faust program; in this case the custom version will override the default one.

28.6 Examples

In the examples subdirectory you'll find a bunch of sample Faust DSPs and Pd patches illustrating how Faust units are used in Pd.

- The examples/basic/test.pd patch demonstrates the basics of operating "bare" Faust plugins in Pd. You'll rarely have to do this when using the wrappers generated with the faust2pd program, but it is a useful starting point to take a look behind the scenes anyway.
- The examples/faust directory contains all the examples from (an earlier version of) the Faust distribution, along with corresponding Pd wrappers generated with faust2pd. Have a look at examples/faust/faustdemo.pd to see some of the DSPs in action. Note that not all examples from the Faust distribution are working out of the box because of name clashes with Pd builtins, so we renamed those. We also edited some of the .dsp sources (e.g., turning buttons into checkboxes or sliders into nentries) where this seemed necessary to make it easier to operate the Pd patches.

- The examples/synth directory contains various plugins and patches showing how to implement polyphonic synthesizers using Faust units. Take a look at examples/synth/synth.pd for an example. If you have properly configured your interfaces then you should be able to play the synthesizer via Pd's MIDI input.
- The examples/seqdemo/seqdemo.pd patch demonstrates how to operate a multitimbral synth, built with Faust units, in an automatic fashion using a pattern sequencer programmed in Pure. This example requires the Pure interpreter as well as the pd-pure plugin available from http://pure-lang.googlecode.com.

28.7 Wrapping DSPs with faust2pd

The faust2pd script generates Pd patches from the dsp.xml files created by Faust when run with the -xml option. Most of the sample patches were actually created that way. After installation you can run the script as follows:

```
faust2pd [-hVbs] [-f size] [-o output-file] [-n #voices]
  [-r max] [-X patterns] [-x width] [-y height] input-file
```

The default output filename is `input-file` with new extension `.pd`. Thus, `faust2pd filename.dsp.xml` creates a Pd patch named `filename.pd` from the Faust XML description in `filename.dsp.xml`.

The faust2pd program understands a number of options which affect the layout of the GUI elements and the contents of the generated patch. Here is a brief list of the available options:

`-h, --help` display a short help message and exit

`-V, --version` display the version number and exit

`-b, --fake-buttons` replace buttons (bangs) with checkboxes (toggles)

`-f, --font-size` font size for GUI elements (10 by default)

`-n, --nvoices` create a synth patch with the given number of voices

`-o, --output-file` output file name (.pd file)

`-r, --radio-sliders` radio controls for sliders

`-s, --slider-nums` sliders with additional number control

`-X, --exclude` exclude controls matching the given glob patterns

`-x, --width` maximum width of the GUI area

`-y, --height` maximum height of the GUI area

Just like the Faust plugin itself, the generated patch has a control input/output as the leftmost inlet/outlet pair, and the remaining plugs are signal inlets and outlets for each audio input/output of the Faust unit. However, the control inlet/outlet pair works slightly different from that of the Faust plugin. Instead of being used for control replies, the control outlet of the patch simply passes through its control input (after processing messages which are

understood by the wrapped plugin). By these means control messages can flow along with the audio signal through an entire chain of Faust units. (You can find an example of this in examples/faust/faustdemo.pd.) Moreover, when generating a polyphonic synth patch using the -n option then there will actually be two control inlets, one for note messages and one for ordinary control messages; this is illustrated in the examples/synth/synth.pd example.

The generated patch also includes the necessary GUI elements to see and change all (active and passive) controls of the Faust unit. Faust control elements are mapped to Pd GUI elements in an obvious fashion, following the horizontal and vertical layout specified in the Faust source. The script also adds special buttons for resetting all controls to their defaults and to operate the special `active` control.

This generally works very well, but you should be aware that the control GUIs generated by faust2pd are somewhat hampered by the limited range of GUI elements available in a vanilla Pd installation. As a remedy, faust2pd provides various options to change the content of the generated wrapper and work around these limitations.

- There are no real button widgets as required by the Faust specification, so bangs are used instead. There is a global delay time for switching the control from 1 back to 0, which can be changed by sending a value in milliseconds to the `faust-delay` receiver. If you need interactive control over the switching time then it is better to use checkboxes instead, or you can have faust2pd automatically substitute checkboxes for all buttons in a patch by invoking it with the -f a.k.a. –fake-buttons option.
- Sliders in Pd do not display their value in numeric form so it may be hard to figure out what the current value is. Therefore faust2pd has an option -s a.k.a. –slider-nums which causes it to add a number box to each slider control. (This flag also applies to Faust's passive bargraph controls, as these are implemented using sliders, see below.)
- Pd's sliders also have no provision for specifying a stepsize, so they are an awkward way to input integral values from a small range. OTOH, Faust doesn't support the "radio" control elements which Pd provides for that purpose. As a remedy, faust2pd allows you to specify the option -r max (a.k.a. –radio-sliders=max) to indicate that sliders with integral values from the range 0..max-1 are to be mapped to corresponding Pd radio controls.
- Faust's bargraphs are emulated using sliders. Note that these are passive controls which just display a value computed by the Faust unit. A different background color is used for these widgets so that you can distinguish them from the ordinary (active) slider controls. The values shown in passive controls are sampled every 40 ms by default. You can change this value by sending an appropriate message to the global `faust-timer` receiver.
- Since Pd has no "tabbed" (notebook-like) GUI element, Faust's tgroups are mapped to hgroups instead. It may be difficult to present large and complicated control interfaces without tabbed dialogs, though. As a remedy, you can control the amount of horizontal or vertical space available for the GUI area with the -x and -y (a.k.a. –width and –height) options and faust2pd will then try to break rows and columns in the layout to make everything fit within that area. (This feature has only been tested with simple layouts so far, so beware.)

- You can also exclude certain controls from appearing in the GUI using the -X option. This option takes a comma-separated list of shell glob patterns indicating either just the names or the fully qualified paths of Faust controls which are to be excluded from the GUI. For instance, the option `-X 'volume,meter*,faust/resonator?/*'` will exclude all volume controls, all controls whose names start with `meter`, and all controls in groups matching `faust/resonator?`. (Note that the argument to -X has to be quoted if it contains any wildcards such as * and `?`, so that the shell doesn't try to expand the patterns beforehand. Also note that only one -X option is recognized, so you have to specify all controls to be excluded as a single option.)
- Faust group labels are not shown at all, since I haven't found an easy way to draw some kind of labelled frame in Pd yet.

Despite these limitations, faust2pd appears to work rather well, at least for the kind of DSPs found in the Faust distribution. (Still, for more complicated control surfaces and interfaces to be used on stage you'll probably have to edit the generated GUI layouts by hand.)

For convenience, all the content-related command line options mentioned above can also be specified in the Faust source, as special tags in the label of the toplevel group of the dsp. These take the form `[pd:option]` or `[pd:option=value]` where `option` is any of the (long) options understood by faust2pd. For instance:

```
process = vgroup("mysynth [pd:nvoices=8] [pd:slider-nums]", ...);
```

Source options carrying arguments, like `nvoices` in the above example, can also be overridden with corresponding command line options.

28.8 Conclusion

Creating Faust plugins for use with Pd has never been easier before, so I hope that you'll soon have much joy trying your Faust programs in Pd. Add Pd-Pure to this, and you can program all your specialized audio and control objects using two modern-style functional languages which are much more fun than C/C++. Of course there's an initial learning curve to be mastered, but IMHO it is well worth the effort. The bottomline is that Pd+Faust+Pure really makes an excellent combo which provides you with a powerful, programmable interactive environment for creating advanced computer music and multimedia applications with ease.

28.8.1 Acknowledgements

Thanks are due to Yann Orlarey for his wonderful Faust, which makes developing DSP algorithms so easy and fun.

28.9 Appendix: faustxml

The faustxml module is provided along with faust2pd to retrieve the description of a Faust DSP from its XML file as a data structure which is ready to be processed by Pure programs. It may also be useful in other Pure applications which need to inspect description of Faust DSPs.

The main entry point is the `info` function which takes the name of a Faust-generated XML file as argument and returns a tuple `(name, descr, version, in, out, controls)` with the name, description, version, number of inputs and outputs and the toplevel group with the descriptions of the controls of the dsp. A couple of other convenience functions are provided to deal with the control descriptions.

28.9.1 Usage

Use the following declaration to import this module in your programs:

```
using faustxml;
```

For convenience, you can also use the following to get access to the module's namespace:

```
using namespace faustxml;
```

28.9.2 Data Structure

The following constructors are used to represent the UI controls of Faust DSPs:

constructor `faustxml::`**`button`** label
constructor `faustxml::`**`checkbox`** label
A button or checkbox with the given label.

constructor `faustxml::`**`nentry`** (label,init,min,max,step)
constructor `faustxml::`**`vslider`** (label,init,min,max,step)
constructor `faustxml::`**`hslider`** (label,init,min,max,step)
A numeric input control with the given label, initial value, range and stepwidth.

constructor `faustxml::`**`vbargraph`** (label,min,max)
constructor `faustxml::`**`hbargraph`** (label,min,max)
A numeric output control with the given label and range.

constructor `faustxml::`**`vgroup`** (label,controls)
constructor `faustxml::`**`hgroup`** (label,controls)
constructor `faustxml::`**`tgroup`** (label,controls)
A group with the given label and list of controls in the group.

28.9.3 Operations

`faustxml::`**`controlp`** x
: Check for control description values.

`faustxml::`**`control_type`** x
`faustxml::`**`control_label`** x
`faustxml::`**`control_args`** x
: Access functions for the various components of a control description.

`faustxml::`**`controls`** x
: This function returns a flat representation of a control group x as a list of basic control descriptions, which provides a quick way to access all the control values of a Faust DSP. The grouping controls themselves are omitted. You can pass the last component of the return value of the `info` function to this function.

`faustxml::`**`pcontrols`** x
: Works like the `controls` function above, but also replaces the label of each basic control with a fully qualified path consisting of all control labels leading up to the given control. Thus, e.g., the label of a slider `"gain"` inside a group `"voice#0"` inside the main `"faust"` group will be denoted by the label `"faust/voice#0/gain"`.

`faustxml::`**`info`** fname
: Extract the description of a Faust DSP from its XML file. This is the main entry point. Returns a tuple with the name, description and version of the DSP, as well as the number of inputs and outputs and the toplevel group with all the control descriptions. Raises an exception if the XML file doesn't exist or contains invalid contents.

Example:

```
> using faustxml;
> let name,descr,version,in,out,group =
>   faustxml::info "examples/basic/freeverb.dsp.xml";
> name,descr,version,in,out;
"freeverb","freeverb -- a Schroeder reverb","1.0",2,2
> using system;
> do (puts.str) $ faustxml::pcontrols group;
faustxml::hslider ("freeverb/damp",0.5,0.0,1.0,0.025)
faustxml::hslider ("freeverb/roomsize",0.5,0.0,1.0,0.025)
faustxml::hslider ("freeverb/wet",0.3333,0.0,1.0,0.025)
```

Chapter 29

pd-faust

Version 0.4, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

pd-faust is a dynamic environment for running Faust dsps in Pd. It is based on the author's *faust2pd* script, but offers many small improvements and some major additional features:

- Faust dsps are implemented using two Pd objects, `fsynth~` and `fdsp~`, which provide the necessary infrastructure to run Faust synthesizer and effect units in Pd, respectively.
- In contrast to faust2pd, the Pd GUI of Faust units is generated dynamically, inside Pd. While pd-faust supports the same global GUI layout options as faust2pd, it also provides various options to adjust the layout of individual control items.
- pd-faust recognizes the `midi` and `osc` controller attributes in the Faust source and automatically provides corresponding MIDI and OSC controller mappings. OSC-based controller automation is also available.
- Perhaps most importantly, Faust dsps can be reloaded at any time (even while the Pd patch is running), in which case the GUI and the controller mappings are regenerated automatically and on the fly as needed.

29.1 Copying

Copyright (c) 2011 by Albert Graef

pd-faust is distributed under the GNU LGPL v3+. Please see the included COPYING and COPYING.LESSER files for details.

This package also includes the faust-stk instruments which are distributed under an MIT-style license, please check the examples/dsp/README-STK file and the dsp files for au-

thorship information and licensing details pertaining to these. The original faust-stk sources can be found in the Faust distribution, cf. http://faust.grame.fr/.

29.2 Installation

You'll need Faust and Pd, obviously. Fairly recent versions of these are required. Faust versions 0.9.46 and 2.0.a3 and Pd version 0.43.1 have been tested.

The pd-faust objects are written in the Pure programming language, so you'll also need an installation of the Pure interpreter (0.51 or later), along with the following packages (minimum required versions are given in parentheses): *pd-pure* (0.15), *pure-faust* (0.8), *pure-midi* (0.5) and *pure-stldict* (0.3).

Finally, gcc and GNU make (or compatible) are required to compile the helper dsps and the example instruments; please check the Makefile for details.

For a basic installation run `make`, then `sudo make install`. This will install the pd-faust objects in your lib/pd/extra/faust folder. Add this directory to your Pd library search path (`-path` option or `Preferences/Path` in Pd) and you should be set. The `make` command also compiles the Faust dsps included in the distribution, so that the provided examples will be ready to run afterwards as well (see Examples below).

The Makefile tries to guess the installation prefix under which Pd is installed. If it guesses wrong, you can tell it the right prefix with `make prefix=/some/path`. Or you can specify the exact path of the lib/pd directory with `make pdlibdir=/some/path`; by default the Makefile assumes `$(prefix)/lib/pd`.

It is also possible to specify an alternative flavour of Pd when building and installing the module, by adding a definition like `PD=pd-extended` to the `make` command line. This is known to work with pd-extended and pd-l2ork, two popular alternative Pd distributions available on the web.

The pd-faust objects are installed both in source form (so you can customize them for your own purposes) and in binary form as a Pd object library which can be loaded with Pd's `-lib` option. The latter substantially reduces startup times and is thus the recommended way to run pd-faust if you don't need to customize the pd-faust objects on the fly. To these ends, after completing installation and setting up `Preferences/Path` in Pd, simply add `pdfaust` to your preloaded library modules in Pd's `Preferences/Startup` dialog.

Note: The `pdfaust` module must come *after* the `pure` entry which loads pd-pure, otherwise you'll get an error message. In any case the pd-pure loader will be required to run these objects, so it should be configured accordingly; please check the *pd-pure* documentation for details.

Some further build options are described in the Makefile. In particular, it is possible to compile the Faust dsps to LLVM bitcode which can be loaded directly by the Pure interpreter, but for that you'll need a special Faust version (see the Faust2 website for how to get this

version up and running) and an LLVM-capable C/C++ compiler such as clang or gcc with the dragonegg plugin (please check the Makefile and the LLVM website for details).

If you have the required tools then you can build the bitcode modules by running `make bitcode` after `make`. If you run `make install` afterwards, the bitcode modules will be installed along with the "normal" Faust plugins. In addition, a second object library called `pdfaust2` will be built and installed, which can be used as a drop-in replacement for `pdfaust` and lets you run the bitcode modules. (Note that in the present implementation it is not possible to load both `pdfaust` and `pdfaust2` in Pd, you'll have to pick one or the other.)

29.3 Usage

Working with pd-faust basically involves adding a bunch of `fsynth~` and `fdsp~` objects to a Pd patch along with the corresponding GUI subpatches, and wiring up the Faust units in some variation of a synth-effects chain which typically takes input from Pd's MIDI interface (`notein`, `ctlin`, etc.) and outputs the signals produced by the Faust units to Pd's audio interface (`dac~`).

For convenience, pd-faust also includes the `midiseq` and `oscseq` objects and a corresponding `midiosc` abstraction which can be used to handle MIDI input and playback as well as OSC controller automation. This useful helper abstraction is described in more detail under Operating the Patches below.

Note: pd-faust interprets MIDI, OSC and Faust dsp filenames relative to the hosting Pd patch by default. It will also search the `midi`, `osc` and `dsp` subfolders, if they exist, for the corresponding types of files. Failing that, it finally searches the directories on the Pd library path (including their `midi`, `osc` and `dsp` subfolders). To disable this search, just use absolute pathnames (or pathnames relative to the `.` or `..` directory) instead.

29.3.1 The fdsp~ and fsynth~ Objects

The `fdsp~` object is invoked as follows:

```
fdsp~ dspname instname channel
```

- `dspname` denotes the name of the Faust dsp (usually this is just the name of the .dsp file with the extension stripped off). Please note that the Faust dsp must be provided in a form which can be loaded in *Pure* (not Pd!), so the `pure.cpp` architecture included in recent Faust versions must be used to compile the dsp to a shared library. (If you're already running Faust2, you can also compile to an LLVM bitcode file instead; Pure has built-in support for loading these.) The Makefiles included in the pd-faust distribution show how to do this.
- `instname` denotes the name of the instance of the Faust unit. Multiple instances of the same Faust dsp can be used in a Pd patch, which must all have different instance

names. In addition, the instance name is also used to identify the GUI subpatch of the unit (see below) and to generate unique OSC addresses for the unit's control elements.

- `channel` is the number of the MIDI channel the unit responds to. This can be 1..16, or 0 to specify "omni" operation (listen to MIDI messages on all channels).

Note: Since the `fdsp~` and `fsynth~` objects are written in Pure, their creation arguments should be specified in Pure syntax. In particular, both `dspname` or `instname` may either be Pure identifiers or double-quoted strings (the former will automatically be translated to the latter). Similarly, the `channel` argument (as well as the `numvoices` argument of the `fsynth~` object, see below) must be an integer constant in Pure syntax, which is pretty much like Pd syntax but also allows the integer to be specified in hexadecimal, octal or binary.

The `fdsp~` object requires a Faust dsp which can work as an effect unit, processing audio input and producing audio output. The unit can have as many audio input and output channels as you like (including zero).

The `fsynth~` object works in a similar fashion, but has an additional creation argument specifying the desired number of voices:

```
fsynth~ dspname instname channel numvoices
```

The `fsynth~` object requires a Faust dsp which can work as a monophonic synthesizer. This typically means that the unit has zero audio inputs and a nonzero number of audio outputs, although it is possible to have synths processing any number of audio input channels as well. (You can even have synths producing zero audio outputs, but this is generally not very useful.) In addition, pd-faust assumes that the Faust unit provides three so-called "voice controls" which indicate which note to play:

- `freq` is the fundamental frequency of the note in Hz.
- `gain` is the velocity of the note, as a normalized value between 0 and 1. This usually controls the volume of the output signal.
- `gate` indicates whether a note is currently playing. This value is either 0 (no note to play) or 1 (play a note), and usually triggers the envelop function (ADSR or similar).

pd-faust doesn't care at which path inside the Faust dsp these controls are located, but they must all be there, and the basenames of the controls must be unique throughout the entire dsp. Otherwise the synth will not work as expected.

Like *faust2pd*, pd-faust implements the necessary logic to drive the given number of voices of an `fsynth~` object. That is, it will actually create a separate instance of the Faust dsp for each voice and handle polyphony by allocating voices from this pool in a round-robin fashion, performing the usual voice stealing if the number of simultaneous notes to play exceeds the number of voices. Also note that an `fsynth~` operated in omni mode (`channel = 0`) automatically filters out messages on channel 10 which is reserved for percussion in the General MIDI standard.

The `fdsp~` and `fsynth~` objects respond to the following messages:

- `bang` outputs the current control settings on the control outlet in OSC format.
- `write` outputs the current control settings to external MIDI and/or OSC devices. This message can also be invoked with a numeric argument to toggle the "write mode" of the unit; please see External MIDI and OSC Controllers below for details.
- `reload` reloads the Faust unit. This also reloads the shared library or bitcode file if the unit was recompiled since the object was last loaded. (Instead of feeding a `reload` message to the control inlet of a Faust unit, you can also just send a `bang` to the `reload` receiver.)
- `addr value` changes the control indicated by the OSC address `addr`. This is also used internally for communication with the Pd GUI and for controller automation.

In addition, the `fdsp~` and `fsynth~` objects respond to MIDI controller messages of the form `ctl val num chan`, and the `fsynth~` object also understands note-related messages of the form `note num vel chan` (note on/off) and `bend val chan` (pitch bend). In either case, pd-faust provides the necessary logic to map controller and note-related messages to the corresponding control changes in the Faust unit.

Note: Like pd-pure, pd-faust also remaps Pd's `menu-open` command so that it lets you edit the Faust source of an `fdsp~` or `fsynth~` object by right-clicking on the object and choosing `Open` from the context menu.

29.3.2 GUI Subpatches

For each `fdsp~` and `fsynth~` object, the Pd patch should also contain an (initially empty) "one-off" graph-on-parent subpatch with the same name as the instance name of the Faust unit:

```
pd instname
```

You shouldn't insert anything into this subpatch, its contents (a bunch of Pd GUI elements corresponding to the control elements of the Faust unit) will be generated automatically by pd-faust when the corresponding `fdsp~` or `fsynth~` object is created, and whenever the unit gets reloaded at runtime.

As with faust2pd, the default appearance of the GUI can be adjusted in various ways; see Tweaking the GUI Layout below for details.

The relative order in which you insert an `fdsp~` or `fsynth~` object and its GUI subpatch into the main patch matters. Normally, the GUI subpatch should be inserted *first*, so that it will be updated automatically when its associated Faust unit is first created, and also when the main patch is saved and then reloaded later.

However, in some situations it may be preferable to insert the GUI subpatch *after* its associated Faust unit. If you do this, the GUI will *not* be updated automatically when the main patch is loaded, so you'll have to reload the dsp manually (sending it a `reload` message) to

force an update of the GUI subpatch. This is useful, in particular, if you'd like to edit the GUI patch manually after it has been generated.

In some cases it may even be desirable to completely "lock down" the GUI subpatch. This can be done by simply *renaming* the GUI subpatch after it has been generated. When Pd saves the main patch, it saves the current status of the GUI subpatches along with it, so that the renamed subpatch will remain static and will *never* be updated, even if its associated Faust unit gets reloaded. This generally makes sense only if the control interface of the Faust unit isn't changed after locking down its GUI patch. To "unlock" a GUI subpatch, you just rename it back to its original name. (In this case you might also want to reinsert the corresponding Faust unit afterwards, if you want to have the GUI generated automatically without an explicit `reload` again.)

29.3.3 Examples

The examples folder contains a few example patches which illustrate how this all works. Having installed pd-faust as described above, you can run these from the examples directory, e.g.: `pd test.pd`. (You can also run the examples without actually installing pd-faust if you invoke Pd from the main pd-faust source directory, e.g., as follows: `pd -lib lib/pdfaust examples/test.pd`.)

Here are some of the examples that are currently available:

- test.pd: Simple patch running a single Faust instrument.
- synth.pd: Slightly more elaborate patch featuring a synth-effects chain.
- bouree.pd: Full-featured example running various instruments.

For your convenience, related MIDI and OSC files as well as the Faust sources of the instruments and effects are contained in corresponding subdirectories (midi, osc, dsp) of the examples directory. A slightly modified version of the faust-stk instruments from the Faust distribution is also included, please check the examples/dsp/README-STK file for more information about these.

The MIDI files are all in standard MIDI file format. (Some of these come from the faust-stk distribution, others can be found on the web.) The OSC files used by pd-faust for controller automation are plain ASCII files suitable for hand-editing if you know what you are doing; the format should be fairly self-explanatory.

29.3.4 Operating the Patches

The generated Pd GUI elements for the Faust dsps are pretty much the same as with *faust2pd* (which see). The only obvious change is the addition of a "record" button (gray toggle in the upper right corner) which enables recording of OSC automation data.

In each example distributed with pd-faust you can also find an instance of the `midiosc` abstraction which serves as a little sequencer applet that enables you to control MIDI playback

and OSC recording. The usage of this abstraction should be fairly obvious, but you can also find a brief description below.

Note: If you use the `midiosc` abstraction in your own patches, you should copy it to the directory containing your patch and other required files, so that MIDI and OSC files are properly located. Alternatively, you can also set up Pd's search path as described at the beginning of the Usage section.

The first creation argument of `midiosc` is the name of the MIDI file, either as a Pure identifier (in this case the .mid filename extension is supplied automatically) or as a double-quoted string. Similarly, the second argument specifies the name of the OSC file. Both arguments are optional; if the second argument is omitted, it defaults to the name of the MIDI file with new extension .osc. You can also omit both arguments if neither MIDI file playback nor saving recorded OSC data is required. Or you can leave the first parameter empty (specify `""` or `0` instead) to only set an OSC filename, if you don't need MIDI playback. The latter is useful, in particular, if you use `midiosc` with an external MIDI sequencer (see below).

The abstraction has a single control outlet through which it feeds the generated MIDI and other messages to the connected `fsynth~` and `fdsp~` objects. Live MIDI input is also accepted and forwarded to the control outlet, after being translated to the format understood by `fsynth~` and `fdsp~` objects. In addition, `midiosc` can also be controlled through an external MIDI sequencer connected to Pd's MIDI input. To these ends, MIDI Machine Control (MMC) can be used to start and stop OSC playback and recording with the transport controls of the external sequencer program. To make this work, the external sequencer must be configured as an MMC master.

At the bottom of the abstraction there is a little progress bar along with a time display which indicates the current song position. If playback is stopped, you can also use these to change the current position for playback, recording and a number of other operations as described below. Note that if you drive `midiosc` from an external MIDI sequencer instead, then it is a good idea to load the same MIDI file in `midiosc` anyway, so that it knows about the length of the MIDI sequence. This will make the progress bar display the proper position in the file.

Here is a brief rundown of the available controls:

- The `start`, `stop` and `cont` controls in the *first* row of control elements start, stop and continue MIDI and OSC playback, respectively. The `echo` toggle in this row causes played MIDI events to be printed in the Pd main window.
- The gray "record" toggle in the upper right corner of the abstraction enables recording of OSC controller automation data. Note that this toggle merely *arms* the OSC recorder; you still have to actually start the recording with the `start` button. However, you can also first start playback with `start` and then switch recording on and off as needed at any point in the sequence (this is also known as "punch in/out" recording). In either case, pushing the `stop` button stores the recorded sequence for later playback. Also note that before you can start recording any OSC data, you first have to arm the Faust units that you want to record. This is done with the "record" toggle in the Pd GUI of each unit.

- The "bang" button next to the "record" toggle lets you record a static snapshot of the current parameter settings of all armed units. This is also done automatically when starting a fresh recording. The "bang" button lets you change the starting defaults of parameters of an existing recording. It is also useful if you just want to record a static snapshot of the current parameter settings without recording any live parameter changes. Moreover, you can also set the parameters at any given point in the piece if you first position the progress bar or the time display accordingly; in this case you may first want to recall the parameter settings at the given point with the `send` button described below. In either case, recording must be enabled and playback must be *off*. Then just arm the Faust units that you wish to record, set the playback position as needed, change the controls to what you want their values to be (maybe after recalling the current settings), and finally push the "bang" button.
- There are some additional controls related to OSC recording in the *second* row: `save` saves the currently recorded data in an OSC file for later use; `abort` is like `stop` in that it stops recording and playback, but also throws away the data recorded in this take (rather than keeping it for later playback); and `clear` purges the entire recorded OSC sequence so that you can start a new one.
- Once some automation data has been recorded, it will be played back along with the MIDI file. You can then just listen to it, or go on to record more automation data as needed. Use the `echo` toggle in the second row to print the OSC messages as they are played back. If you save the automation data with the `save` button, it will be reloaded from its OSC file next time the patch is opened.
- The controls in the *third* row provide some additional ways to configure the playback process. The `loop` button can be used to enable looping, which repeats the playback of the MIDI (and OSC) sequence ad infinitum. The `thru` button, when switched on, routes the MIDI data during playback through Pd's MIDI output so that it can be used to drive an external MIDI device in addition to the Faust instruments. The `write` button does the same with MIDI and OSC controller data generated either through automation data or by manually operating the control elements in the Pd GUI, see External MIDI and OSC Controllers below for details.
- There's one additional button in the third row, the `send` button which recalls the recorded OSC parameter settings at a given point in the sequence. Playback must be off for this to work. After setting the playback position as desired, just push the `send` button. This sets the controls to the current parameter values at the given point, for *all* parameters which have been recorded up to (and including) this point.

Please note that `midiosc` is merely an example which should cover most common uses. If you don't need all the fancy functionality it provides, your patches may simply feed control messages directly into `fdsp~` and `fsynth~` objects instead. On the other hand, if you need even more elaborate input/output interfacing than what `midiosc` provides, `midiosc` can be used as a starting point for your own input/output abstractions driving the `fdsp~` and `fsynth~` objects in your patch.

29.3.5 External MIDI and OSC Controllers

The `fsynth~` object has built-in (and hard-wired) support for MIDI notes, pitch bend and MIDI controller 123 (all notes off).

Other controller data received from external MIDI and OSC devices is interpreted according to the controller mappings defined in the Faust source (this is explained below), by updating the corresponding GUI elements and the control variables of the Faust dsp. For obvious reasons, this only works with *active* Faust controls.

An `fdsp~` or `fsynth~` object can also be put in *write mode* by feeding a message of the form `write 1` into its control inlet (the `write 0` message disables write mode again). For convenience, the `write` toggle in the `midiosc` abstraction allows you to do this simultaneously for all Faust units connected to `midiosc`'s control outlet.

When an object is in write mode, it also *outputs* MIDI and OSC controller data in response to both automation data and the manual operation of the Pd GUI elements, again according to the controller mappings defined in the Faust source, so that it can drive an external device such as a MIDI fader box or a multitouch OSC controller. Note that this works with both *active* and *passive* Faust controls.

To configure MIDI controller assignments, the labels of the Faust control elements have to be marked up with the special `midi` attribute in the Faust source. For instance, a pan control (MIDI controller 10) may be implemented in the Faust source as follows:

```
pan = hslider("pan [midi:ctrl 10]", 0, -1, 1, 0.01);
```

pd-faust will then provide the necessary logic to handle MIDI input from controller 10 by changing the pan control in the Faust unit accordingly, mapping the controller values 0..127 to the range and step size given in the Faust source. Moreover, in write mode corresponding MIDI controller data will be generated and sent to Pd's MIDI output, on the MIDI channel specified in the creation arguments of the Faust unit (0 meaning "omni", i.e., output on all MIDI channels).

The same functionality is also available for external OSC devices, employing explicit OSC controller assignments in the Faust source by means of the `osc` attribute. E.g., the following enables input and output of OSC messages for the OSC `/pan` address:

```
pan = hslider("pan [osc:/pan]", 0, -1, 1, 0.01);
```

Note: In contrast to some other architectures included in the Faust distribution, pd-faust only allows literal OSC addresses here. That is, glob-style OSC patterns are not supported as values for the `osc` attribute.

Also note that while the necessary infrastructure to support OSC input and output is already provided by pd-faust, the OSC input and output facilities themselves which are needed to make this actually work are not implemented in the default version of the `midiosc` abstraction distributed with pd-faust. That's because Pd doesn't provide any built-in objects for

OSC input and output, although a few different solutions exist and are included in Hans-Christoph Steiner's Pd-extended distribution, such as Martin Peach's OSC externals.

The distribution includes a version of the `midiosc` abstraction which implements OSC input and output using Martin Peach's objects in the midiosc-mrpeach.pd file. You most likely have to edit this abstraction for your purposes; at least you'll have to change the network addresses so that it works with the OSC device or application that you use.

29.3.6 Tweaking the GUI Layout

As already mentioned, pd-faust provides the same global GUI layout options as *faust2pd*. Please check the faust2pd documentation for details. There are a few minor changes in the meaning of some of the options, though, which we consider notable improvements after some experience working with faust2pd. Here is a brief rundown of the available options, as they are implemented in pd-faust:

- `width=wd`, `height=ht`: Specify the maximum horizontal and/or vertical dimensions of the layout area. If one or both of these values are nonzero, pd-faust will try to make the GUI fit within this area.
- `font-size=sz`: Specify the font size (default is 10).
- `fake-buttons`: Render `button` controls as Pd toggles rather than bangs.
- `radio-sliders=max`: Render sliders with up to `max` different values as Pd radio controls rather than Pd sliders. Note that in pd-faust this option not only applies to sliders, but also to numeric entries, i.e., `nentry` in the Faust source. However, as with faust2pd's `radio-sliders` option, the option is only applicable if the control is zero-based and has a stepsize of 1.
- `slider-nums`: Add a number box to each slider control. Note that in pd-faust this is actually the default, which can be disabled with the `no-slider-nums` option.
- `exclude=pat,...`: Exclude the controls whose labels match the given glob patterns from the Pd GUI.

In pd-faust there is no way to specify the above options on the command line, so you'll have to put them as `pd` attributes on the *main* group of your Faust program, as described in the faust2pd documentation. For instance:

```
process = vgroup("[pd:no-slider-nums][pd:font-size=12]", ...);
```

In addition, the following options can be used to change the appearance of individual control items. If present, these options override the corresponding defaults. Each option can also be prefixed with "`no-`" to negate the option value. (Thus, e.g., `no-hidden` makes items visible which would otherwise, by means of the global `exclude` option, be removed from the GUI.)

- `hidden`: Hides the corresponding control in the Pd GUI. This is the only option which can also be used for group controls, in which case *all* controls in the group will become invisible in the Pd GUI.

- `fake-button`, `radio-slider`, `slider-num`: These have the same meaning as the corresponding global options, but apply to individual control items.

Again, these options are specified with the `pd` attribute in the label of the corresponding Faust control. For instance, the following Faust code hides the controls in the `aux` group, removes the number entry from the `pan` control, and renders the `preset` item as a Pd radio control:

```
aux = vgroup("aux [pd:hidden]", aux_part);
pan = hslider("pan [pd:no-slider-num]", 0, -1, 1, 0.01);
preset = nentry("preset [pd:radio-slider]", 0, 0, 7, 1);
```

29.3.7 Remote Control

Also included in the sources is a helper abstraction faust-remote.pd and an accompanying elisp program faust-remote.el. These work pretty much like pure-remote.pd and pure-remote.el in the *pd-pure* distribution, but are tailored for the remote control of Faust dsps in a Pd patch. In particular, they enable you to quickly reload the Faust dsps in Pd using a simple keyboard command (`C-C C-X` by default) from Emacs. The faust-remote.el program was designed to be used with Juan Romero's Emacs Faust mode; please see etc/pure-remote.el in the pd-faust source for usage instructions.

29.4 Caveats and Bugs

Some parts of this software might still be experimental, under construction and/or bug-ridden. Bug reports, patches and suggestions are welcome. Please send these directly to the author, or post them either to the Faust or the Pure mailing list.

In particular, please note the following known limitations in the current implementation:

- Passive Faust controls are only supported in `fdsp~` objects.
- The names of the voice controls in the `fsynth~` object (`freq`, `gain`, `gate`) are currently hard-coded, as are the names of the `midi`, `osc` and `dsp` subfolders used to locate various kinds of files.
- Polyphonic aftertouch and channel pressure messages are not supported in the MIDI interface right now, so you'll have to use ordinary MIDI controllers for these parameters instead. Coarse/fine pairs of MIDI controllers aren't directly supported either, so you'll have to implement these yourself as two separate Faust controls.

Also, please check the TODO file included in the distribution for other issues which we are already aware of and which will hopefully be addressed in future pd-faust versions.

Chapter 30

pd-pure: Pd loader for Pure scripts

Version 0.16, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This is a Pd loader plugin for the Pure programming language which lets you write external Pd objects in Pure.

Please note that the present version of the module is still somewhat experimental, but it seems to work fairly well at least with Pure versions 0.34 and later. In particular, note that Pure is a *compiled* language and thus there are some inevitable latencies at startup, when the embedded Pure interpreter loads your Pure scripts and compiles them on the fly. However, once the scripts have been compiled, they are executed very efficiently. As of pd-pure 0.15, it is also possible to *precompile* a collection of Pure objects to a library in binary format which can be loaded much faster with Pd's `-lib` option. (This requires Pure 0.50 or later.)

30.1 Copying

Copyright (c) 2009-2011 by Albert Graef. pd-pure is distributed under a 3-clause BSD-style license, please see the included COPYING file for details.

30.2 Installation

MS Windows users please see pd-pure on Windows below.

Get the latest source from http://pure-lang.googlecode.com/files/pd-pure-0.16.tar.gz.

Usually, `make && sudo make install` should do the trick. This will compile the external (you need to have GNU make, Pd and Pure installed to do that) and install it in the lib/pd/extra/pure directory.

The Makefile tries to guess the installation prefix under which Pd is installed. If it guesses wrong, you can tell it the right prefix with `make prefix=/some/path`. Or you can specify the exact path of the lib/pd directory with `make pdlibdir=/some/path`; by default the Makefile assumes `$(prefix)/lib/pd`.

It is also possible to specify an alternative flavour of Pd when building and installing the module, by adding a definition like `PD=pd-extended` to the `make` command line. This is known to work with pd-extended and pd-l2ork, two popular alternative Pd distributions available on the web.

The Makefile also tries to guess the host system type and Pure version, and set up some platform-specific things accordingly. If this doesn't work for your system then you'll have to edit the Makefile accordingly.

30.2.1 pd-pure on Windows

There's a binary package in MSI format available at the Pure website: http://pure-lang.googlecode.com/files/pd-pure-0.16.msi. Use that if you can. You'll also need the latest Pure version (0.50 at the time of this writing), and Pd 0.43 or later, which is available from Miller Puckette's website: http://crca.ucsd.edu/~msp/software.html.

30.3 Usage

After installation, you still have to tell Pd to load the Pure external at startup, either with the `-lib` option (`pd -lib pure`), or by specifying `pure` in the File/Startup options (Media/Preferences/Startup in Pd 0.43 and later). This setting can be saved so that the Pure loader is always available when you run Pd. Once the Pure loader has been installed, you should see a message in the Pd main window indicating that the external has been loaded.

Since version 0.12 pd-pure supports the definition of both *control* and *audio objects* in Pure. The latter are also known as "tilde objects" in Pd parlance; pd-pure follows the Pd convention in that audio objects have a trailing tilde in their name. Audio objects are used primarily for processing audio signals, whereas control objects are employed for asynchronous message processing.

Simple "one-off" control objects can be created with the `[pure]` class which takes the function to be evaluated as its argument. For instance:

```
[pure (+5)]
```

This object takes numbers as inputs on its single inlet, adds 5 to them and outputs the result on its single outlet.

Similarly, audio objects can be created with `[pure~]`. For instance, the following object processes incoming vectors of samples, multiplying each sample with 2:

```
[pure~ map (*2)]
```

Note that in this case the object has actually two inlet/outlet pairs. The leftmost inlet/outlet pair is reserved for the processing of control messages (not used in this example), while the actual signal input and output can be found on the right.

(Pure objects can also be configured to adjust the number of inlets and outlets. This will be described later.)

The argument of `[pure]` and `[pure~]` can be any Pure expression (including local functions and variables, conditionals, etc.). We also refer to these as *anonymous* Pure objects. If an object is quite complicated or used several times in a patch, it is more convenient to implement it as a *named* object instead. To these ends, the object function is stored in a corresponding Pure script named after the object. For instance, we might put the following `add` function into a script named add.pure:

```
add x y = x+y;
```

Now we can use the following object in a Pd patch:

```
[add 5]
```

The Pure loader then recognizes `add` as an instance of the object implemented by the add.pure file and loads the script into the Pure interpreter. The creation parameter `5` is passed as the first argument `x` of the `add` function in this example, while the `y` argument comes from the object's inlet. The function performed by this object is thus the same as with `[pure (+5)]` above.

More examples can be found in the pure-help.pd and pure~-help.pd patches. These can also be accessed in Pd by right-clicking on any Pure object and selecting the `Help` option. (Recent pd-pure versions also allow you to right-click and select `Open` to open the script of a named Pure object in a text editor, provided that your Pd version supports the `menu-open` command.)

In the following section, we first discuss in detail how control objects are defined and used. After that, the necessary adjustments for implementing audio objects are explained. Some advanced uses of pd-pure are described under Advanced Features.

30.4 Control Objects

Basically, to implement a Pd control object named `foo`, all you have to do is supply a Pure script named foo.pure which defines a function `foo` (and anything else that you might need to define the function). This function is also called the *object function*. You can put the script containing the object function in the same directory as the Pd patch in which you want to use the `foo` object, or anywhere on Pd's search path. (The latter is useful if the object is to be used in several patches located in different subdirectories.) The script will be executed once, at the time the first object with the given name is created, and will be executed in the directory where it is located. Thus, if the script needs to import other Pure scripts or load

some data files, you can put these into the same directory so that the object script can find them.

The `foo` function gets evaluated at object creation time, receiving any additional parameters the object is created with. The resulting Pure expression should be another function which is executed at runtime, passing Pd messages from the inlets as parameters, and routing the function results to the outlets of the object. This two-stage definition process is useful because it allows special processing (such as initialization of required data structures) to be done at object creation time. However, the result of evaluating `foo` can also just be `foo` itself if no such special processing is needed. If we need to distinguish these two stages, we also call the two functions the *creation* and the *runtime* function of the object, respectively.

Pd messages are translated to corresponding Pure expressions and vice versa in a straightforward fashion. Special support is provided for converting between the natural Pd and Pure representations of floating point numbers, symbols and lists. The following table summarizes the available conversions.

Message Type	Pd	Pure
symbol	`foo`	`foo`
string	`a&b`	`"a&b"`
float	`float 1.23`	`1.23`
list	`list 1 2 3`	`[1.0,2.0,3.0]`
other	`foo a 2 3`	`foo a 2.0 3.0`

Note that Pd symbols which are no valid Pure symbols become strings in Pure. Conversely, both symbols and strings in Pure are mapped to corresponding Pd symbols. Pure (machine) integers and floating point values both become `float` messages in Pd. Pd list messages are translated to Pure list values, while other aggregate messages are mapped to Pure applications (and vice versa).

30.4.1 Simple Objects

By default, a Pure object has just one inlet and one outlet and thus acts like a simple function with no internal state. For instance, the following object accepts Pd `float` messages and adds 5 to each received value:

```
add5 x = x+5;
```

In the Pd patch each `[add5]` object then has a single inlet supplying parameters and a single outlet for results of the add5 function.

30.4.2 Creation Arguments

You can parameterize an object with creation arguments, which are passed to the Pure function at object creation time. For instance:

```
add x y = x+y;
```

This object can then be invoked, e.g., as `[add 5]` in the Pd patch to supply the needed creation argument `x`.

30.4.3 The [pure] Object

For simple kinds of objects like the above, the Pure loader provides the generic `[pure]` object as a quick means to create Pure control objects without actually preparing a script file. The creation parameter of `[pure]` is the object function. This can be a predefined Pure function, or you can define it on the fly in a `with` clause. (It is also possible to explicitly load additional script files needed to implement objects defined using `[pure]`; see Controlling the Runtime for details.)

For instance, `[pure succ]` uses the predefined Pure function `succ` which adds 1 to its input, while the object `[pure add 5 with add x y = x+y end]` produces the same results as the `[add 5]` object defined using a separate add.pure script in the previous section. You can also generate constant values that way. E.g., the object `[pure cst 1.618]` responds to any message (such as `bang`) by producing the constant value 1.618, while the object `[pure cst [1..10]]` yields the constant list containing the numbers 1..10.

30.4.4 Configuring Inlets and Outlets

To create an object with multiple inlets and outlets for control messages, the object creation function must return the desired numbers of inlets and outlets, along with a second function to be applied at runtime, as a tuple `n,m,foo`. The input arguments to the runtime function as well as the corresponding function results are then encoded as pairs `k,val` where `k` denotes the inlet or outlet index. (Note that the `k` index is provided only if there actually is more than one inlet. Also, the outlet index is assumed to be zero if none is specified, so that it can be omitted if there's only one outlet.)

For instance, the following object, invoked as `[cross]` in the Pd patch, has two inlets and two outlets and routes messages from the left inlet to the right outlet and vice versa:

```
cross = 2,2,cross with cross (k,x) = (1-k,x) end;
```

You can also emit multiple messages, possibly to different outlets, in one go. These must be encoded as Pure vectors (or matrices) of values or `index,value` pairs, which are emitted in the order in which they are written. E.g., the following `[fan]` object implements an "n-fan" which routes its input to `n` outlets simultaneously:

```
fan n = 1,n,fan with fan x = reverse {k,x | k = 0..n-1} end;
```

(Note that, because of the use of `reverse`, the `n` outlets are served in right-to-left order here. This is not strictly necessary, but matches the Pd convention.)

Another example is the following `[dup]` object with a single inlet and outlet, which just sends out each received message twice:

```
dup x = {x,x};
```

Note that this is different from the following, which outputs a list value to the outlet instead:

```
dup2 x = [x,x];
```

(Also, please note that this behaviour is new in pd-pure 0.14. Previously, a list return value by itself would output multiple values instead. However, this made it very awkward to deal with Pd list values in pd-pure, and so as of pd-pure 0.14 Pure matrices must now be used to output multiple values.)

An object can also just "swallow" messages and generate no output at all. To these ends, make the object return either an empty vector `{}` or the empty tuple `()`. (Note that, in contrast, returning the empty list `[]` does send a value back to Pd, namely an empty list value.) For instance, the following object `[echo]` implements a sink which just prints received messages on standard output, which is useful for debugging purposes:

```
using system;
echo x = () when puts (str x) end;
```

You could also implement this object as follows, by just removing the superflous outlet (in this case all return values from the function will be ignored anyway):

```
using system;
echo = 1,0,puts.str;
```

30.4.5 Local State

Local state can be kept in Pure reference values. For instance, the following `[counter]` object produces the next counter value when receiving a `bang` message:

```
nonfix bang;
counter = next (ref 0) with
  next r bang = put r (get r+1);
  next _ _    = () otherwise;
end;
```

Note that the state is kept as an additional first parameter to the local function `next` here. Alternatively, you can also make the state a local variable of `counter`:

```
nonfix bang;
counter = next with
  next bang = put r (get r+1);
  next _    = () otherwise;
end when r = ref 0 end;
```

30.5 Audio Objects

If the name of a Pure object (i.e., the basename of the corresponding Pure script) ends with the ~ character, pd-pure assumes that it denotes an audio object whose primary purpose is to process sample data. The basic setup is similar to the case of control objects, with the following differences:

- The object function for an audio object `xyz~` is named `xyz_dsp` (rather than `xyz`). The function is defined in the `xyz~.pure` script file, which must be located in the same directory as the Pd patch or anywhere on Pd's search path.
- To keep things simple, a Pure audio object is always equipped with exactly one control inlet and one control outlet, which are the leftmost inlet and outlet of the object. These can be used to process control messages in the usual fashion, in addition to the audio processing performed by the object.
- Any additional inlets and outlets of the object are signal inlets and outlets. By default, one signal inlet/outlet pair will be provided. Configuring a custom number of signal inlets and outlets works as with control objects. In this case the object creation function must return a triple `n,m,foo` where `n` and `m` are the desired number of signal inlets and outlets, respectively, and `foo` is the actual processing function to be invoked at runtime.

Whenever Pd has audio processing enabled, the object function is invoked with one block of sample data for each iteration of Pd's audio loop. The sample data is encoded as a double matrix which has one row for each signal inlet of the object; row 0 holds the sample data for the first signal inlet, row 1 the sample data for the second signal inlet, etc. The row size corresponds to Pd's *block size* which indicates how many samples per signal connection is processed in one go for each iteration of the audio loop. (Usually the default block size is 64, but this can be changed with Pd's `-blocksize` option and also on a per-window basis using the `block~` object, see the Pd documentation for details.) Note that the input matrix will have zero rows if the object has zero signal inlets, in which case the row size of the matrix (as reported by the `dim` function) still indicates the block size.

When invoked with a signal matrix as argument, the object function should return another double matrix with the resulting sample data for the signal outlets of the object, which normally has one row per outlet and the same row size as the input matrix. (A lack or surplus of samples in the output matrix is handled gracefully, however. Missing samples are filled with zeros, while extra samples are silently ignored.)

For instance, here's a simple object with the default single signal inlet/outlet pair (in addition to the leftmost control inlet/outlet pair, which isn't used in this example). This object just multiplies its input signal by 2:

```
mul2_dsp x::matrix = map (*2) x;
```

This code would then be placed into a script file named `mul2~.dsp` and invoked in Pd as an object of the form `[mul2~]`.

As with control objects, there's a shortcut to create simple objects like these without preparing a script file, using the built-in `[pure~]` object. Thus the dsp function in the previous

example could also be implemented using an object of the form `[pure~ map (*2)]` (which uses the same function, albeit in curried form).

Creation parameters can also be used in the same way as with control objects. The following object is to be invoked in Pd as `[mul~ f]` where `f` is the desired gain factor.

```
mul_dsp f::double x::matrix = map (*f) x;
```

Next, let's try a custom number of signal inlets and outlets. The following object has two signal inlets and one signal outlet. Like Pd's built-in `[*~]` object, it multiplies the two input signals, producing an amplitude (or ring) modulation effect:

```
sigmul_dsp = 2,1,sigmul with
  sigmul x::matrix = zipwith (*) (row x 0) (row x 1);
end;
```

Here's another example which takes no inputs and produces one output signal, a random wave (i.e., white noise). Note the use of the `dim` function to determine the number of samples to be generated for each block.

```
extern double genrand_real1() = random1;
randomwave1_dsp = 0,1,randomwave with
  randomwave in::matrix = {random | i=1..n} when _,n = dim in end;
  random = random1*2-1;
end;
```

Control messages for the control outlet of the object may be added by returning a pair `sig,msg` where `sig` is the output signal matrix and `msg` is a single control message or vector of such messages (using the same format as with control objects). The signal matrix can also be omitted if no signal output is needed (unless the control data takes the form of a double matrix, which would be interpreted as signal data; in such a case you'd have to specify an empty signal matrix instead). The object function may also return `()` if neither signal nor control output is required. (This may be the case, e.g., for dsps which just analyze the incoming signal data and store the results somewhere for later retrieval.)

Audio objects can also process control messages and generate responses on the leftmost inlet/outlet pair as usual. This is commonly used to set and retrieve various control parameters used or generated by the audio processing part of the object.

For instance, here is an audio object which plays back a soundfile using the `sndfile` module (cf. *pure-audio*). The object function reads the entire file (whose name is passed as a creation argument) at creation time and turns over processing to the `playsf` function which returns one block of samples from the file (along with the current position of the playback pointer) for each invocation with an (empty) input matrix. In addition, a `bang` message is output when the end of the file is reached. The object also responds to floating point values in the range from 0 to 1 on the control inlet by adjusting the playback pointer accordingly.

```
using sndfile;

nonfix bang;
```

```
playsf_dsp name = 0,nchannels,playsf with
  // Play one block of samples. Also output a number in the range 0..1 on the
  // control outlet to indicate the current position.
  playsf x::matrix = block,get pos/nsamples when
    _,n = dim x; block = submat buf (0,get pos) (nchannels,n);
    put pos (get pos+n);
  end if get pos>=0 && get pos<=nsamples;
  // Output a bang once to indicate that we're done.
  playsf x::matrix = bang when
    _,n = dim x; put pos (-1);
  end if get pos>=0;
  playsf _::matrix = ();
  // A number in the range 0..1 places the playback pointer accordingly.
  playsf x::double = put pos $ int $ round $ x*nsamples $$ ();
end when
  // Open the audio file for reading.
  info = sf_info (); sf = sf_open name SFM_READ info;
  // Get some information about the file.
  nsamples,rate,nchannels,_ = sf_get_info info;
  nsamples = int nsamples;
  // Read the file into memory.
  buf = dmatrix (nsamples,nchannels);
  nsamples = int $ sf_readf_double sf buf nsamples;
  // Convert interleaved samples (nsamples x nchannels) to one channel per row
  // (nchannels x nsamples).
  buf = transpose buf;
  // Initialize the playback pointer:
  pos = ref 0;
end;
```

As another example, here's a complete stereo amplifier stage with bass, treble, gain and balance controls and a dB meter. The dsp part is implemented in Faust, Grame's functional dsp programming language. The Pure program just does the necessary interfacing to Pd, which includes processing of incoming control messages for setting the control parameters of the Faust dsp, and the generation of output control messages to send the dB meter values (also computed in the Faust dsp) to Pd. (To run this example, you need the "faust2" branch of the Faust compiler so that the dsp can be inlined into the Pure program. Note that the entire section inside the `%< %>` braces is Faust code.)

```
%< -*- dsp:amp -*-

import("math.lib");
import("music.lib");

/* Fixed bass and treble frequencies. You might want to tune these for your
   setup. */

bass_freq       = 300;
treble_freq     = 1200;

/* Bass and treble gain controls in dB. The range of +/-20 corresponds to a
```

```
   boost/cut factor of 10. */

bass_gain       = nentry("bass", 0, -20, 20, 0.1);
treble_gain     = nentry("treble", 0, -20, 20, 0.1);

/* Gain and balance controls. */

gain            = db2linear(nentry("gain", 0, -96, 96, 0.1));
bal             = hslider("balance", 0, -1, 1, 0.001);

/* Balance a stereo signal by attenuating the left channel if balance is on
   the right and vice versa. I found that a linear control works best here. */

balance         = *(1-max(0,bal)), *(1-max(0,0-bal));

/* Generic biquad filter. */

filter(b0,b1,b2,a0,a1,a2)       = f : (+ ~ g)
with {
        f(x)    = (b0/a0)*x+(b1/a0)*x'+(b2/a0)*x'';
        g(y)    = 0-(a1/a0)*y-(a2/a0)*y';
};

/* Low and high shelf filters, straight from Robert Bristow-Johnson's "Audio
   EQ Cookbook", see http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt. f0
   is the shelf midpoint frequency, g the desired gain in dB. S is the shelf
   slope parameter, we always set that to 1 here. */

low_shelf(f0,g)         = filter(b0,b1,b2,a0,a1,a2)
with {
        S  = 1;
        A  = pow(10,g/40);
        w0 = 2*PI*f0/SR;
        alpha = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 );

        b0 =    A*( (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha );
        b1 =  2*A*( (A-1) - (A+1)*cos(w0)                   );
        b2 =    A*( (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha );
        a0 =        (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha;
        a1 =   -2*( (A-1) + (A+1)*cos(w0)                   );
        a2 =        (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha;
};

high_shelf(f0,g)        = filter(b0,b1,b2,a0,a1,a2)
with {
        S  = 1;
        A  = pow(10,g/40);
        w0 = 2*PI*f0/SR;
        alpha = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 );

        b0 =    A*( (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha );
        b1 = -2*A*( (A-1) + (A+1)*cos(w0)                   );
```

```
        b2 =    A*( (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha );
        a0 =        (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha;
        a1 =    2*( (A-1) - (A+1)*cos(w0)                   );
        a2 =        (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha;
};

/* The tone control. We simply run a low and a high shelf in series here. */

tone            = low_shelf(bass_freq,bass_gain)
                : high_shelf(treble_freq,treble_gain);

/* Envelop follower. This is basically a 1 pole LP with configurable attack/
   release time. The result is converted to dB. You have to set the desired
   attack/release time in seconds using the t parameter below. */

t               = 0.1;                  // attack/release time in seconds
g               = exp(-1/(SR*t));       // corresponding gain factor

env             = abs : *(1-g) : + ~ *(g) : linear2db;

/* Use this if you want the RMS instead. Note that this doesn't really
   calculate an RMS value (you'd need an FIR for that), but in practice our
   simple 1 pole IIR filter works just as well. */

rms             = sqr : *(1-g) : + ~ *(g) : sqrt : linear2db;
sqr(x)          = x*x;

/* The dB meters for left and right channel. These are passive controls. */

left_meter(x)   = attach(x, env(x) : hbargraph("left", -96, 10));
right_meter(x)  = attach(x, env(x) : hbargraph("right", -96, 10));

/* The main program of the Faust dsp. */

process         = (tone, tone) : (_*gain, _*gain) : balance
                : (left_meter, right_meter);
%>

// These are provided by the Pd runtime.
extern float sys_getsr(), int sys_getblksize();
// Provide some reasonable default values in case the above are missing.
sys_getsr = 48000; sys_getblksize = 64;

// Get Pd's default sample rate and block size.
const SR = int sys_getsr;
const n = sys_getblksize;

using faustui;

amp_dsp = k,l,amp with
  // The dsp part. This also outputs the left and right dbmeter values for
  // each processed block of samples on the control outlet, using messages of
```

```
  // the form left <value> and right <value>, respectively.
  amp in::matrix = amp::compute dsp n in out $$
    out,{left (get_control left_meter),right (get_control right_meter)};
  // Respond to control messages of the form <control> <value>. <control> may
  // be any of the input controls supported by the Faust program (bass,
  // treble, gain, etc.).
  amp (c@_ x::double) = put_control (ui!str c) x $$ x;
end when
  // Initialize the dsp.
  dsp = amp::newinit SR;
  // Get the number of inputs and outputs and the control variables.
  k,l,ui = amp::info dsp;
  ui = control_map $ controls ui;
  {left_meter,right_meter} = ui!!["left","right"];
  // Create a buffer large enough to hold the output from the dsp.
  out = dmatrix (l,n);
end;
```

Note that it is possible to load the above Faust program directly in Pd, using the facilities described in *faust2pd: Pd Patch Generator for Faust*. This is also more efficient since it avoids the overhead of the extra Pure layer. However, invoking Faust dsps via Pure also offers some benefits. In particular, it enables you to add more sophisticated control processing, interface to other 3rd party software for additional pre- and postprocessing of the signal data, or do live editing of Faust programs using the facilities described in Livecoding below.

30.6 Advanced Features

This section discusses some advanced features of the Pd Pure loader. It explains the use of timer callbacks, wireless connections and wave arrays, and the livecoding and interactive control facilities. We also give an overview of the API provided for pd-pure programmers.

30.6.1 Asynchronous Messages

pd-pure provides a simple asynchronous messaging facility which allows a Pure object to schedule a message to be delivered to itself later. This is useful for implementing all kinds of delays and, more generally, any kind of object which, once triggered, does its own sequencing of control messages.

To these ends, the object function may return a special message of the form `pd_delay t msg` (either by itself or as an element of a result list) to indicate that the message `msg` should be delivered to the object function after `t` milliseconds (where `t` is either a machine int or a double value). After the prescribed delay the object function will then be invoked on the given message, and the results of this call are processed as usual (routing messages to outlets and/or scheduling new timer events in response to further `pd_delay` messages). Note that if the delay is zero or negative, the message is scheduled to be delivered immediately.

For instance, a simple kind of delay object can be implemented in Pure as follows:

```
mydelay _ (alarm msg) = msg;
mydelay t msg = pd_delay t (alarm msg) otherwise;
```

The desired delay time is specified as a creation argument. The first equation handles messages of the form `alarm msg`; the action is to just output the delayed message given by the `msg` argument. All other input messages are scheduled by the second equation, which wraps the message in an `alarm` term so that it gets processed by the first equation when it is delivered.

Note that pd-pure only allows you to schedule a single asynchronous event per call of the object function. Thus, if the `mydelay` object above receives another message while it is still waiting for the previous one to be delivered, the old timer is cancelled and the new one is scheduled instead; this works like Pd's builtin `delay` object.

Moreover, scheduling a new event at an infinite (or `nan`) time value cancels any existing timer. (Note that you still have to specify the `msg` parameter, but it will be ignored in this case.) We can use this to equip our `mydelay` object with a `stop` message as follows:

```
nonfix stop;
mydelay _ (alarm msg) = msg;
mydelay _ stop = pd_delay inf ();
mydelay t msg = pd_delay t (alarm msg) otherwise;
```

More elaborate functionality can be built on top of the basic timer facility. The following example shows how to maintain a timed message queue in a Pure list, in order to implement a simple delay line similar to Pd's builtin `pipe` object. Here we also employ the `pd_time()` function, which is provided by the Pure loader so that Pure scripts can access the current logical Pd time in milliseconds (see Programming Interface below). This is convenient if we need to deal with absolute time values, which we use in this example to keep track of the times at which messages in the queue are to be delivered:

```
extern double pd_time();
mypipe t = process (ref []) with
  process q () = case dequeue q of
                   x,(t,_):_ = {x,pd_delay (t-pd_time) ()};
                   x,_ = x;
                 end;
  process q x  = enqueue q x $$ pd_delay t () if null (get q);
               = enqueue q x $$ () otherwise;
  enqueue q x  = put q $ get q+[(pd_time+t,x)];
  dequeue q    = x,put q xs when (_,x):xs = get q end;
end;
```

30.6.2 Wireless Messaging

As of version 0.14, pd-pure offers some facilities for sending and receiving messages directly, without any wired connections to the inlets and outlets of an object (similar to what the Pd `[send]` and `[receive]` objects provide). See the description for the `pd_send()` and `pd_receive()` routines in the Programming Interface section.

For instance, here's how you can use the `pd_send` function to send messages to the Pd runtime:

```
pd_send "pd" (dsp 1); // turn on audio processing
```

This function also enables you to perform dynamic patching, by sending the appropriate messages to patches (i.e., `"pd-patch"` receivers, where `patch` is the name of the target patch). Useful messages to patches are listed in the Tips and Tricks section on the Pd community website, and some examples can be found here. For instance, the following Pure object, when banged, inserts a few objects into a subpatch named `test` and connects them to each other:

```
extern void pd_send(char*, expr*);

pd_send_test _ = () when
  pd_send "pd-test" (obj 10 0 "osc~" 220);
  pd_send "pd-test" (obj 10 30 "*~" 0.1);
  pd_send "pd-test" (obj 10 60 "dac~");
  pd_send "pd-test" (connect 0 0 1 0);
  pd_send "pd-test" (connect 1 0 2 0);
  pd_send "pd-test" (connect 1 0 2 1);
end;
```

An object can also receive messages from any named source by means of the `pd_receive` function. This function must be called either at object creation time or when one of the dsp or control processing methods of the object is invoked. For instance, the following object calls `pd_receive` at creation time in order to receive messages sent to the `left` and `right` receivers, and outputs them on its left or right outlet, respectively:

```
extern void pd_receive(char*);

pd_receive_test = 1,2,process with
  process (left x) = 0,x;
  process (right x) = 1,x;
end when
  do pd_receive ["left","right"];
end;
```

Please note that `pd_receive` itself doesn't return any message, it merely registers a receiver symbol so that messages sent to that symbol may be received later. The received messages are always delivered to the leftmost inlet when Pd does its control processing. Moreover, the symbol identifying the source of the message is applied to the message itself, so that the receiver can figure out where the message came from and adjust accordingly. This operation is useful, in particular, to provide communication channels between Pd GUI elements and Pure objects. Wireless connections are often preferred in such cases, to reduce display clutter.

30.6.3 Reading and Writing Audio Data

Besides the realtime processing of audio data, Pd also provides a means to store sample data in *arrays* which can be displayed in a patch and modified interactively, see the section on

numeric arrays in the Pd documentation for details. Arrays can be used, e.g., as running waveform displays, as wavetables which are played back in the audio loop, or as waveshaping functions used to implement distortion effects.

Each array has a name (Pd symbol) under which it can be accessed from Pure code. pd-pure makes it possible to transfer audio data directly between Pd arrays and Pure double vectors by means of the `pd_getbuffer()` and `pd_setbuffer()` routines. Please see Programming Interface below for a closer description of the provided routines.

For instance, here is a `randomwave` object which fills a Pd array (whose name is given as the creation argument) with random values in response to a `bang` message:

```
extern double genrand_real1() = random1;

extern int pd_getbuffersize(char *name);
extern void pd_setbuffer(char *name, expr *x);

nonfix bang;

randomwave name = 1,0,process with
  process bang  = pd_setbuffer name {random | i = 1..nsamples};
  nsamples      = pd_getbuffersize name;
  random        = random1*2-1;
end;
```

30.6.4 Controlling the Runtime

pd-pure provides a predefined `[pure-runtime]` object which makes it possible to control the embedded Pure interpreter in some ways. There can be any number of `[pure-runtime]` objects in a patch, which all refer to the same instance of the Pure interpreter.

The first use of `[pure-runtime]` is to load additional Pure scripts. To these ends, `[pure-runtime]` can be invoked with the names of scripts to be loaded at object creation time as arguments. The script names should be specified without the `.pure` suffix; it will be added automatically. The scripts will be searched for in the directory of the patch containing the `[pure-runtime]` object and on the Pd path. For instance, to load the scripts `foo.pure` and `bar.pure`, you can add the following object to your patch:

```
[pure-runtime foo bar]
```

This facility can be used, e.g., to load any additional scripts needed for anonymous objects defined with `[pure]` and `[pure~]`.

Note: You'll have to make sure that the `[pure-runtime]` object is inserted into the patch before any anonymous objects which depend on the loaded scripts. Also note that you shouldn't explicitly load the scripts which implement named Pure objects; this will be handled automatically by the Pure loader.

The `[pure-runtime]` object also accepts control messages which can be used to dynamically reload all loaded scripts, and to implement "remote control" of a patch using the **pdsend** program. This is described in the following subsection.

30.6.5 Livecoding

Livecoding means changing Pure objects on the fly while a patch is running. A simple, but limited way to do this is to just edit the boxes containing Pure objects interactively, as you can do with any kind of Pd object. In this case, the changes take effect immediately after you finish editing a box. However, for more elaborate changes, you may have to edit the underlying Pure scripts and notify the Pure interpreter so that it reloads the scripts. The Pure loader provides the special `[pure-runtime]` object to do this.

Sending a `bang` to the `[pure-runtime]` object tells the plugin to reload all object scripts and update the Pure objects in your patch accordingly. The object also provides two outlets to deal with the inevitable latencies caused by the compilation process. The right outlet is banged when the compilation starts and the left outlet gets a bang when the compilation is finished, so that a patch using this facility can respond to these events in the appropriate way (e.g., disabling output during compilation).

The `reload` message works similarly, but while the `bang` message only reloads the object scripts, `reload` restarts the Pure interpreter from scratch and reloads everything, including the prelude and imported modules. This will usually take much longer, but is only necessary if you edited imported library modules which won't be reloaded with the `bang` message.

While this facility is very useful for interactive development, it does have some limitations:

- At present, the number of inlets and outlets of Pure objects never changes after reloading scripts. Pd does not support this through its API right now. Thus by editing and reloading the Pure scripts you can change the functionality of existing Pure objects in a running patch, but not their interfaces. (It is possible to make changes to inlets and outlets take effect by manually editing the affected objects afterwards. But this will be cumbersome when you have to edit a lot of objects, so it might be easier to just restart Pd and reload the patches in such cases.)
- Reloading scripts may need some time depending on how much Pure code has to be recompiled and how fast your cpu is. With the `bang` message the delays will usually be small, but still noticable. In order to keep the compilation times to a minimum, it is a good idea to put all code which you don't plan to edit "live" into library modules which are imported in the object scripts. By these means, the number of definitions in the object scripts themselves can be kept small, resulting in faster compilation.

Also note that the reloading of object scripts amounts to a "cold restart" of the Pure objects in your patches. If a Pure object keeps some local state, it will be lost. As a remedy, the loader implements a simple protocol which allows Pure objects to record their internal state before a script gets reloaded, and restore it afterwards. To these ends, a Pure object may respond to the following two messages:

- Before reloading, the Pure object will receive the `pd_save` message. In response, the

object should return a Pure expression encoding its internal state in a way which can be serialized (see the description of the `blob` function in the *Pure Library Manual* for details). Usually, it is sufficient to just pack up all state data in a tuple, list or some other aggregate and return that as the response to the `pd_save` message.

- After reloading, the Pure object will receive a `pd_restore state` message, where `state` is the previously recorded state, as returned by the object in response to the `pd_save` message. It should then restore its internal state from the saved data. (The return value of this message invocation is ignored.)

In order to participate in the `pd_save/pd_restore` protocol, an object must subscribe to it. This is done by setting `pd_save` as a sentry on the object function (see the description of the `sentry` function in the *Pure Library Manual* for details). For instance, here's the counter example from Local State again, with the necessary additions to support the `pd_save/pd_restore` protocol:

```
nonfix bang pd_save;
counter = sentry pd_save $ next (ref 0) with
  next r bang = put r (get r+1);
  next r pd_save = get r;
  next r (pd_restore n) = put r n;
  next _ _ = () otherwise;
end;
```

30.6.6 Remote Control

The distribution also includes an abstraction pure-remote.pd which you can include in your patch to enable live coding, as well as remote control of the patch through the **pdsend** program. Sending a `bang` (or `reload`) causes a reload of the object scripts, as described above. This can also be triggered directly by clicking the bang control of the abstraction. The bang control also provides visual feedback indicating whether the compilation is still in progress. Messages are routed through the embedded `[pure-runtime]` object using the single inlet and the two outlets of the abstraction, so that `pure-remote` can also be controlled from within the patch itself.

For added convenience, the `[pure-runtime]` and `[pure-remote]` objects also accept any other message of the form `receiver message` and will route the given message to the given receiver. This is intended to provide remote control of various parameters in patches. For instance, by having **pdsend** send a `play 0` or `play 1` message, one might implement a simple playback control, provided that your patch includes an appropriate receiver (often a GUI object). See the pure-help.pd patch for an example.

To make these features available in **emacs**, there's an accompanying elisp program (pure-remote.el) which contains some convenient keybindings for the necessary **pdsend** invocations, so that you can operate the pure-remote patch with simple keystrokes directly from the text editor. The same bindings are also available in Emacs Pure mode, but need to be enabled before you can use them; please see the pure-remote.el file for details. As shipped, pure-remote.el and Pure mode implement the following commands:

C-C C-X	Quick Reload	Sends a bang message to reload object scripts.
C-C M-X	Full Reload	Sends a reload message to reload everything.
C-C C-M	Message	Prompts for a message and sends it to pure-remote.
C-C C-S	Play	Sends a play 1 message.
C-C C-T	Stop	Sends a play 0 message.
C-C C-G	Restart	Sends a play 0 message followed by play 1.
C-/	Dsp On	Sends a pd dsp 1 (enable audio processing).
C-.	Dsp Off	Sends a pd dsp 0 (disable audio processing).

Of course you can easily add more like these, just have a look at how the keybindings are implemented in pure-remote.el or pure-mode.el and create your own in an analogous fashion. Together with Pure mode, this gives you a nice interactive environment for developing pd-pure applications.

30.6.7 Compiling Objects

pd-pure's livecoding abilities require that objects are run from source code. As already mentioned, this needs some (in some cases, substantial) time at startup when the Pure interpreter is loaded and your Pure scripts are compiled to native code on the fly. This is wasted effort if you are finished developing your Pure objects and just want to run them as they are.

Therefore pd-pure also supports compiling a collection of Pure objects to a binary which can be loaded with Pd's `-lib` option just like any other external library of Pd objects. This basically involves using the Pure interpreter as a batch compiler to translate the Pure scripts implementing the objects to a shared library. You also have to link in a small amount of C code so that the shared module can be loaded by Pd and registers its Pd object classes with pd-pure. The examples/lib folder contains a complete example showing how this is done.

Note that even if you load all your pd-pure objects from such libraries, you still need to load the pd-pure module first, since it provides the basic infrastructure required to run any kind of pd-pure object (no matter whether it's implemented in compiled or source form).

30.6.8 Programming Interface

The Pure loader provides a number of interface routines which can be called by Pure scripts running in the Pd environment. Note that in order to access these functions, you'll have to add the corresponding `extern` declarations to your scripts.

extern char ***pd_version_s**()
: Returns the Pd version number as a string. Note that this routine will only be available when a script is running inside Pd, so you can quickly check if that's the case as follows:

```
let ok = stringp $ eval "extern char *pd_version_s(); pd_version_s;";
```

The `ok` variable will then be true iff the script is running inside Pd.

extern expr ***pd_path**()
: Returns the Pd path (set in Pd's `Path` dialog or via the `-path` command line option) as

a list of directory names. This is useful if your Pure scripts need to locate files on the Pd search path.

extern char ***pd_libdir**()
Returns the Pd library dir (as determined at compile time). This is useful if your Pure scripts need to access files in that directory.

extern expr ***pd_getdir**()
Returns the directory of the patch the current object is in. This is useful if a Pure object needs to access files in the patch directory. Please note that this function must be called during object creation or in the method calls of an object, so that it is clear what the current object is; otherwise the function will fail. Also note that the results may differ for different instances of the same object class, depending on which patches the objects are located in.

extern expr ***pd_getfile**()
Returns the name of the file that will be opened with the `menu-open` action (accessible by right-clicking on a Pure object and selecting `Open`). This is usually the Pure script of the object, if available, but this can be changed with `pd_setfile()` below. The function must be called during object creation or in the method calls of an object, so that it is clear what the current object is; otherwise the function will fail.

extern void **pd_setfile**(char **s*)
Sets the name of the file to be opened with the `menu-open` action. By default, this is the Pure script of the object, if available; this function can be used to change the name of the file on a per-object basis. The function must be called during object creation or in the method calls of an object, so that it is clear what the current object is; otherwise the function will have no effect.

extern void **pd_post**(char **s*)
Posts a message in the Pd main window. A trailing newline is added automatically. This is a convenience function which is equivalent to calling Pd's `post()` (which is a varargs function) as `post "%s" s`.

extern void **pd_error_s**(char **s*)
Like `pd_post()`, but prints an error message instead. If this routine is invoked from an object (i.e., during object creation or a method call) then Pd's `pd_error()` function is called, which allows the object to be tracked down with Pd's `Find Last Error` menu command. Otherwise (i.e., if the function is called at load time) Pd's `error()` function is called which just outputs the message.

extern double **pd_time**()
Retrieves the current Pd time as a double value in milliseconds, which is useful, in particular, when used in conjunction with the asynchronous message facility described under Asynchronous Messages.

extern void **pd_send**(char **sym*, expr **x*)
Sends a message, specified as a Pure term `x`, to the receiver given by the symbol `sym` (specified as a string). This is a no-op if the receiver doesn't exist.

extern void **pd_receive**(char **sym*)

Prepares an object so that it can receive messages sent to the given symbol `sym`. This function must be called during object creation or method calls. It can be called for different symbols, as many times as needed. The messages are always delivered to the leftmost inlet, and the given symbol is applied to the original message, so that the receiver can figure out where the message came from.

extern void **pd_unreceive**(char **sym*)
Switches off receiving messages for the given symbol `sym`. Use this to undo the effects of a previous `pd_receive` call.

extern expr ***pd_getbuffer**(char **name*)
extern void **pd_setbuffer**(char **name*, expr **x*)
extern int **pd_getbuffersize**(char **name*)
extern void **pd_setbuffersize**(char **name*, uint32_t *sz*)
Routines to access the Pd array (sample buffer) with the given name. These functions can be used to transfer audio data between Pd and Pure scripts; see Reading and Writing Audio Data above for an example.

`pd_getbuffersize()` and `pd_setbuffersize()` gets or sets the size of the given buffer, respectively.

`pd_getbuffer()` reads the contents of the buffer and returns it as a Pure vector (or fails if the array with the given name doesn't exist).

`pd_setbuffer()` sets the contents of the buffer from the given Pure vector `x`. If the size of the vector exceeds the size of the buffer, the former is truncated. Conversely, if the size of the buffer exceeds the size of the Pure vector, the trailing samples are unaffected. *Note:* The second argument of `pd_setbuffer()` can also be a pair `(i,x)` denoting an offset `i` into the array at which the sample data is to be written, so that this routine allows you to overwrite any part of the array.

Chapter 31

pure-audio

Version 0.5, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This is a digital audio interface for the Pure programming language. It currently includes the following modules:

- audio.pure: A PortAudio wrapper which gives you portable access to realtime audio input and output on a variety of different host APIs. This uses the well-known PortAudio library by Ross Bencina, Phil Burk et al, see http://www.portaudio.com/.
- fftw.pure: Compute real-valued FFTs of audio signals using Matteo Frigo's and Steven G. Johnson's portable and fast FFTW library ("Fastest Fourier Transform in the West").
- sndfile.pure: Reading and writing audio files in various formats. This is a fairly straightforward wrapper for Erik de Castro Lopo's libsndfile library, see http://www.mega-nerd.com/libsndfile/.
- samplerate.pure: Perform sample rate conversion on audio data. This uses another of Erik's excellent libraries, libsamplerate (a.k.a. SRC), see http://www.mega-nerd.com/SRC/.
- realtime.pure: A little utility module which provides access to realtime scheduling to Pure programs. You may need this for low-latency realtime audio applications.

Documentation still needs to be written, so for the time being please read the source modules listed above and have a look at the examples provided in the distribution.

31.1 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-audio-0.5.tar.gz.

You need to have libportaudio (v19), libsndfile (1.x), libsamplerate (0.1.x) and libfftw3 (3.x) installed on your system. Any fairly recent version of these libraries should do. For the

realtime module you also need a POSIX threads library (libpthread) with the POSIX realtime thread extension; Linux, OSX and other Un*x systems should offer this.

The Pure wrappers contained in the distribution are for 64 bit POSIX systems. If you're running a 32 bit system, or Windows, then you should regenerate them using 'make generate'. This requires the header files portaudio.h, samplerate.h and sndfile.h (and pure-gen, of course). If you do this, check the includedir variables defined in the Makefiles, these need to point to the directories where the corresponding header files are to be found (the default is /usr/include).

Then just run 'make' to compile the package. If you're lucky and everything compiles smoothly, you can install with 'sudo make install'.

If you're not so lucky, you can get help on the Pure mailing list, see http://groups.google.com/group/pure-lang.

31.2 License

pure-audio is Copyright (c) 2010 by Albert Graef, licensed under the 3-clause BSD license, see the COPYING file for details.

Please note that if you're using these modules, you're also bound by the license terms of the PortAudio, libsamplerate and libsndfile libraries they are based on, see the corresponding sources and websites for details.

Chapter 32

pure-faust

Version 0.9, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This module lets you load and run Faust-generated signal processing modules in Pure. Faust (an acronym for Functional AUdio STreams) is a functional programming language for real-time sound processing and synthesis developed at Grame and distributed as GPL'ed software.

Note: As of Pure 0.45, there's also built-in support for Faust interoperability in the Pure core, including the ability to inline Faust code in Pure programs; see *Interfacing to Faust* in the Pure manual. The built-in Faust interface requires Faust2 which is still under development and available as a separate package in the Faust git repository. Both interfaces provide pretty much the same basic capabilities and should work equally well for most applications. In fact, as of version 0.5 pure-faust comes with a compatibility module which provides the pure-faust API on top of the built-in Faust interface, see the description of the `faust2` module below for details.

32.1 Copying

Unless explicitly stated otherwise, this software is Copyright (c) 2009-2012 by Albert Graef. Please also see the source for the copyright and license notes pertaining to individual source files.

pure-faust is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pure-faust is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;

without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

32.2 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-faust-0.9.tar.gz.

Binary packages can be found at http://pure-lang.googlecode.com/. To install from source, run the usual `make && sudo make install`. This requires Pure, of course (the present version will work with Pure 0.52 and later). The Makefile tries to guess the installation prefix under which Pure is installed. If it guesses wrong, you can tell it the right prefix with `make prefix=/some/path`. Or you can specify the exact path of the `lib/pure` directory with `make libdir=/some/path`; by default the Makefile assumes `$(prefix)/lib/pure`. The Makefile also tries to guess the host system type and set up some platform-specific things accordingly. If this doesn't work for your system then you'll have to edit the Makefile accordingly.

The Faust compiler is not required to compile this module, but of course you'll need it to build the examples in the `examples` subdirectory and to compile your own Faust sources. You'll need Faust 0.9.46 or later.

To compile Faust programs for use with this module, you'll also need the `pure.cpp` architecture file. This should be included in recent Faust releases. If your Faust version doesn't have it yet, you can find a suitable version of this file in the `examples` folder. Simply copy the file to your Faust library directory (usually `/usr/local/lib/faust` or similar) or the directory holding the Faust sources to be compiled, and you should be set.

32.3 Usage

Once Faust and this module have been installed as described above, you should be able to compile a Faust dsp to a shared module loadable by pure-faust as follows:

```
$ faust -a pure.cpp -o mydsp.cpp mydsp.dsp
$ g++ -shared -o mydsp.so mydsp.cpp
```

Note that, by default, Faust generates code which does all internal computations with single precision. You can add the `-double` flag to the Faust command in order to use double precision instead. (In either case, all data will be represented as doubles on the Pure side.)

Also note that the above compile command is for a Linux or BSD system using `gcc`. Add `-fPIC` for 64 bit compilation. For Windows compilation, the output filename should be `mydsp.dll` instead of `mydsp.so`; on Mac OSX, it should be `mydsp.dylib`. There's a Makefile in the `examples` folder which automates this process.

Once the module has been compiled, you can fire up the Pure interpreter and load the dsp as follows:

```
> using faust;
> let dsp = faust_init "mydsp" 48000;
> dsp;
#<pointer 0xf09220>
```

The `faust_init` function loads the `"mydsp.so"` module (the `.so` suffix is supplied automatically) and returns a pointer to the Faust dsp object which can then be used in subsequent operations.

Note: `faust_init` only loads the dsp module if it hasn't been loaded before. However, as of pure-faust 0.8, `faust_init` also checks the modification time of the module and reloads it if the module was recompiled since it was last loaded. (This is for compatibility with Pure's built-in Faust interface which behaves in the same way.) If this happens, *all* existing dsp instances created with the old version of the module become invalid immediately (i.e., all subsequent operations on them will fail, except `faust_exit`) and must be recreated.

The second parameter of `faust_init`, 48000 in this example, denotes the sample rate in Hz. This can be an arbitrary integer value which is available to the hosted dsp (it's up to the dsp whether it actually uses this value in some way). The sample rate can also be changed on the fly with the `faust_reinit` function:

```
> faust_reinit dsp 44100;
```

It is also possible to create copies of an existing dsp with the `faust_clone` function, which is quite handy if multiple copies of the same dsp are needed (a case which commonly arises when implementing polyphonic synthesizers):

```
> let dsp2 = faust_clone dsp;
```

When you're done with a dsp, you can invoke the `faust_exit` function to unload it (this also happens automatically when a dsp object is garbage-collected):

```
> faust_exit dsp2;
```

Note that after invoking this operation the dsp pointer becomes invalid and must not be used any more.

In the following, we use the following little Faust program as a running example:

```
declare descr "amplifier";
declare author "Albert Graef";
declare version "1.0";

gain = nentry("gain", 1.0, 0, 10, 0.01);
process = *(gain);
```

The `faust_info` function can be used to determine the number of input/output channels as well as the "UI" (a data structure describing the available control variables) of the loaded

dsp:

```
> let n,m,ui = faust_info dsp;
```

Global metadata of the dsp is available as a list of `key=>val` string pairs with the `faust_meta` function. For instance:

```
> faust_meta dsp;
["descr"=>"amplifier","author"=>"Albert Graef","version"=>"1.0"]
```

To actually run the dsp, you'll need two buffers capable of holding the required number of audio samples for input and output. For convenience, the `faust_compute` routine lets you specify these as Pure double matrices. `faust_compute` is invoked as follows:

```
> faust_compute dsp in out;
```

Here, `in` and `out` must be double matrices which have at least `n` or `m` rows, respectively (corresponding to the number of input and output channels of the Faust dsp). The row size of these matrices determines the number of samples which will be processed (if one of the matrices has a larger row size than the other, the extra elements are ignored). The `out` matrix will be modified in-place and also returned as the result of the call.

Some DSPs (e.g., synthesizers) only take control input without processing any audio input; others (e.g., pitch detectors) might produce just control output without any audio output. In such cases you can just specify an empty `in` or `out` matrix, respectively. For instance:

```
> faust_compute dsp {} out;
```

Most DSPs take additional control input. The control variables are listed in the "UI" component of the `faust_info` return value. For instance, suppose that there's a `gain` parameter listed there, it might look as follows:

```
> controls ui!0;
hslider #<pointer 0x12780a4> [] ("gain",1.0,0.0,10.0,0.1)
```

The constructor itself denotes the type of control, which matches the name of the Faust builtin used to create the control (see the Faust documentation for more details on this). The *third* parameter is a tuple which indicates the arguments the control was created with in the Faust program. The *first* parameter is a C `double*` which points to the current value of the control variable. You can inspect and change this value with the `get_double` and `put_double` routines available in the Pure prelude. (Note that, for compatibility with the internal Faust interface which supports both single and double precision controls, you can also use the `get_control` and `put_control` functions instead.) Changes of control variables only take effect between different invocations of `faust_compute`. Example:

```
> let gain = control_ref (controls ui!0);
> get_double gain;
1.0
> put_double gain 2.0;
()
> faust_compute dsp in out;
```

Output controls such as `hbargraph` and `vbargraph` are handled in a similar fashion, only that the Faust dsp updates these values for each call to `faust_compute` and Pure scripts can then read the values with `get_double` or `get_control`.

The *second* parameter of a control description is a list holding the Faust metadata of the control. This list will be empty if the control does not have any metadata. Otherwise you will find some of `key=>val` string pairs in this list. It is completely up to the application how to interpret the metadata, which may consist, e.g., of GUI layout hints or various kinds of controller definitions. For instance, a MIDI controller assignment might look as follows in the Faust source:

```
gain = nentry("gain[midi:ctrl 7]", 1.0, 0, 10, 0.01);
```

In Pure this information will then be available as:

```
> control_meta (controls ui!0);
["midi"=>"ctrl 7"]
```

Let's finally have a closer look at the contents of the UI data structure. You will find that it is actually a tree, similar to the directory tree of a hierarchical file system, which reflects the layout of the controls in the Faust program. For instance:

```
> ui;
vgroup [] ("mydsp",[nentry #<pointer 0x12780a4> [] ("gain",1.0,0.0,10.0,0.01)])
```

The leaves of the tree are the actual controls, while its interior nodes are so-called "control groups", starting from a root node which represents the entire dsp. There are different kinds of control groups such as `vgroup` and `hgroup`; please check the Faust documentation for details. Control groups have a name and metadata just like individual controls, but there is no `control_ref` component and the data stored at the node is the list of controls and subgroups contained in the control group. The `controls` function returns a flat representation of the controls in the UI tree as a list, omitting the group nodes of the tree:

```
> controls ui;
[hslider #<pointer 0x12780a4> [] ("gain",1.0,0.0,10.0,0.1)]
```

We've already employed this function above to extract the `gain` control of our example dsp. There's a variation of this function which yields the full "pathnames" of controls in the UI tree:

```
> pcontrols ui;
[hslider #<pointer 0x12780a4> [] ("mydsp/gain",1.0,0.0,10.0,0.1)]
```

This is sometimes necessary to distinguish controls with identical names in different control groups. There are two additional convenience functions which work with this flat representation of the UI data structure:

```
> let ctrls = ans;
> control_map ctrls;
{"mydsp/gain"=>#<pointer 0x12780a4>}
> control_metamap ctrls;
{"mydsp/gain"=>[]}
```

The results are Pure records which provide convenient access to the pointers and metadata of the controls by their name.

Please note that, as of Pure 0.45, the UI access functions described above are actually provided by the `faustui` standard library module which gets included by the `faust` module.

Further examples can be found in the examples subdirectory.

32.4 Faust2 Compatibility

As of version 0.5, pure-faust includes a Faust2 compatibility module which lets you use the pure-faust API on top of Pure's new Faust bitcode interface, using the same operations as described under Usage above. This module is invoked with the following import clause:

```
using faust2;
```

To instantiate a Faust dsp using the `faust2` interface, you'll have to compile the Faust program to LLVM bitcode format. The examples directory includes a `pure.c` Faust architecture file to help with this. Please see the *Interfacing to Faust* section in the Pure manual for details.

Note that only one of the `faust` and `faust2` modules may be imported into a program; trying to use both modules in the same program will *not* work. Also note that the `faust2` module requires Faust2 and a fairly recent Pure version to work, whereas the `faust` module works with both Faust2 and the mainline Faust version and doesn't rely on the Faust bitcode loader (only the `pure.cpp` architecture is needed).

32.5 Acknowledgements

Many thanks to Yann Orlarey at Grame, the principal author of Faust!

Chapter 33

pure-liblo

Version 0.8, June 26, 2012

Albert Gräf <Dr.Graef@t-online.de>

33.1 Copying

Copyright (c) 2009 by Albert Graef.

pure-liblo is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

33.2 Description

This is a quick and dirty Pure wrapper for the liblo library by Steve Harris and others, which implements Berkeley's Open Sound Control (OSC) protocol.

OSC is a protocol for exchanging data between multimedia devices and software across the network (TCP, UDP and UNIX domain sockets are supported as the transport layer). It is also useful as a general communication mechanism for both hard- and software. In difference to the plain socket interface (on which it builds), OSC provides you with an efficient means to send around binary data packets along with the corresponding type and timing information, which makes it well-suited for both realtime and non-realtime applications.

The OSC protocol is standardized and is supported by an abundance of different implementations, which includes controller hardware of all sorts and computer music software like CSound, Pd and SuperCollider. Lots of implementations exist for different programming languages. liblo aims to provide a lightweight and ubiquitous OSC implementation for the C programming language.

The `lo.pure` module provides a fairly straight wrapper of the C library. A more high-level and Purified interface is available in `osc.pure`. Most of the time, you'll want to use the latter for convenience, but if you need utmost flexibility then it is worth having a look at `lo.pure`, too.

- Get the latest source from http://pure-lang.googlecode.com/files/pure-liblo-0.8.tar.gz.
- To install, run `make` and `sudo make install`. This will try to guess your Pure installation directory; if it guesses wrong, you can set the `prefix` variable accordingly, see the Makefile for details.
- You can also regenerate the wrapper by running `make generate`; this requires the `pure-gen` utility and the liblo headers. The present version was generated from liblo 0.26. If your liblo version differs from that then it's always a good idea to run `make generate`.
- Have a look at `lo.pure` and `osc.pure` for a description of the API provided to Pure programmers.
- The `examples` folder contains some Pure code which illustrates how to use these modules.

Chapter 34

pure-midi

Version 0.5, June 26, 2012

Albert Graef <Dr.Graef@t-online.de>

This is a MIDI interface for the Pure programming language (Pure 0.45 or later is required). It includes the following modules:

- midi.pure: A PortMidi/PortTime wrapper which gives you portable access to realtime MIDI input and output. This uses PortMidi (by Roger B. Dannenberg et al) from the PortMedia project, see http://portmedia.sourceforge.net/.
- midifile.pure: Reading and writing standard MIDI files. This is based on David G. Slomin's light-weight midifile library, which comes bundled with the pure-midi sources.

Documentation still needs to be written, so for the time being please read the source modules listed above and have a look at the examples provided in the distribution.

34.1 Installation

Get the latest source from http://pure-lang.googlecode.com/files/pure-midi-0.5.tar.gz.

You need to have the PortMidi library installed on your system. This release was tested with PortMidi 2.00 (I recommend using the svn version of PortMidi, since it fixes some 64 bit compilation problems). If you have to use some earlier PortMidi version then you may have to fiddle with portmidi.pure and/or midi.pure to make it work. (You can also just regenerate the wrapper by copying portmidi.h and porttime.h from your PortMidi installation to the pure-midi source directory and running 'make generate'. This requires pure-gen. See the toplevel Makefile for details.)

Run 'make' to compile the package. If you're lucky and everything compiles smoothly, you can install with 'sudo make install'.

If you're not so lucky, you can get help on the Pure mailing list, see http://groups.google.com/group/pure-lang.

NOTE: You may also want to install the related pure-audio package. In particular, pure-audio also provides realtime.pure, a little utility module which gives Pure programs access to realtime scheduling.

34.2 License

pure-midi is Copyright (c) 2010 by Albert Graef, licensed under the 3-clause BSD license, see the COPYING file for details.

For convenience, I've bundled some (BSD-licensed or compatible) source files from other packages with this release. portmidi.h and porttime.h are from PortMidi 2.00 (http://portmedia.sourceforge.net/) which is

Copyright (c) 1999-2000 Ross Bencina and Phil Burk Copyright (c) 2001-2006 Roger B. Dannenberg

midifile.c and midifile.h in the midifile subdirectory are from "Div's midi utilities" (http://public.sreal.com:8000/~div/midi-utilities/) which is

Copyright (c) 2003-2006 David G. Slomin

Please see portmidi.h and midifile.h for the pertaining copyrights and license conditions.

Chapter 35

Installing Pure (and LLVM)

Version 0.56, September 24, 2012

Albert Graef <Dr.Graef at t-online.de>
Eddie Rucker <erucker at bmc.edu>

These instructions explain how to compile and install LLVM (which is the compiler back-end required by Pure) and the Pure interpreter itself. The instructions are somewhat biased towards Linux and other Unix-like systems; the System Notes section at the end of this file details the tweaks necessary to make Pure compile and run on various other platforms. More information about installing LLVM and the required LLVM source packages can be found at http://llvm.org.

Pure is known to work on Linux, FreeBSD, NetBSD, Mac OSX and MS Windows, and should compile (with the usual amount of tweaking) on all recent UNIX/POSIX- based platforms. We recommend using version 4.x of the GNU C++ compiler; it should be available almost everywhere (in fact, since you'll need LLVM anyway, you may also use one of the LLVM-based C/C++ compilers such as llvm-gcc or clang). You'll also need a Bourne-compatible shell and GNU make, which are also readily available on most platforms.

A binary package in msi format is provided for Windows users in the download area at http://pure-lang.googlecode.com. Information about ports and packages for other (UNIX-like) systems is provided at the same location.

35.1 Quick Summary

Here is the executive summary for the impatient. This assumes that you're using LLVM 3.1 and Pure 0.56, please substitute your actual version numbers in the commands given below.

Prerequisites: gcc, GNU make, flex/bison (development sources only), libltdl, libgmp and libmpfr (including header files for development), wget (for downloading and installing the online documentation), GNU emacs (if you want to use Emacs Pure mode). These should all be available as binary packages on most systems.

You'll probably need the latest Pure and LLVM tarballs which, at the time of this writing, are available here:

http://pure-lang.googlecode.com/files/pure-0.56.tar.gz
http://llvm.org/releases/3.1/llvm-3.1.src.tar.gz
http://llvm.org/releases/3.1/clang-3.1.src.tar.gz
http://llvm.org/releases/3.1/dragonegg-3.1.src.tar.gz

Note: If you're reading this documentation online, then the Pure version described here most likely is still under development, in which case you can either grab the latest available release, or install from the development sources instead (see Installing From Development Sources below).

Installing LLVM and clang (the latter is optional but recommended):

```
$ tar xfvz llvm-3.1.src.tar.gz
$ tar xfvz clang-3.1.src.tar.gz && mv clang-3.1.src llvm-3.1.src/tools/clang
$ cd llvm-3.1.src
$ ./configure --enable-shared --enable-optimized --enable-targets=host-only
$ make && sudo make install
```

You may want to leave out `--enable-shared` to install LLVM as static libraries only, and `--enable-targets=host-only` if you want to enable cross compilation for all supported targets in LLVM. (With some older LLVM versions you may also have to add `--disable-assertions --disable-expensive-checks` to disable stuff that makes LLVM very slow and/or breaks it on some systems.)

If you're running gcc 4.5 or later, you may also want to install the LLVM "DragonEgg" plugin for gcc, please check the dragonegg section below for details.

Installing Pure:

```
$ tar xfvz pure-0.56.tar.gz
$ cd pure-0.56
$ ./configure --enable-release
$ make && sudo make install
```

Depending on your system you may have to run a utility to announce the shared LLVM and Pure libraries to the dynamic loader. E.g., on Linux:

```
$ sudo /sbin/ldconfig
```

It is also recommended that you run the following to make sure that the Pure interpreter works correctly on your platform (see step 5 below for details):

```
$ make check
```

The following is optional, but if you want to read the online documentation in the interpreter or in Emacs Pure mode, you'll have to download and install the documentation files:

```
$ sudo make install-docs
```

This needs the `wget` program. You can also download the pure-docs tarball manually and then install it from a local file, e.g.:

```
$ sudo make install-docs docs=pure-docs-0.56.tar.gz
```

That's it, Pure should be ready to go now:

```
$ pure
```

Uninstalling:

```
$ cd pure-0.56
$ sudo make uninstall
$ cd ../llvm-3.1.src
$ sudo make uninstall
```

Please see below for much more detailed installation instructions.

35.2 Basic Installation

The basic installation process is as follows. Note that steps 1-3 are only required once. Steps 2-3 can be avoided if binary LLVM packages are available for your system (but see the caveats about broken LLVM packages on some systems below). Additional instructions for compiling Pure from the latest repository sources can be found in the Installing From Development Sources section below. Moreover, you can refer to the Other Build And Installation Options section below for details about various options available when building and installing Pure.

Step 1. Make sure you have all the necessary dependencies installed (`-dev` denotes corresponding development packages):

- GNU make, GNU C/C++ and the corresponding libraries;
- the GNU multiprecision library (`libgmp`, `-dev`) or some compatible replacement (see comments below);
- the GNU multiprecision floating point library (`libmpfr`, `-dev`);
- GNU readline (`libreadline`, `-dev`) or some compatible replacement (only needed if you want command line editing support in the interpreter; see comments below).
- GNU emacs (if you want to use Pure mode).

In addition, the following will be required to compile the development version (see the Installing From Development Sources section below):

- autoconf;
- flex and bison;
- mercurial (needed to fetch the development sources).

The following may be required to build some LLVM versions:

- GNU ltdl library (`libltdl`, `-dev`).

All dependencies are available as free software. Here are some links if you need or want to install the dependencies from source:

- Autoconf: http://www.gnu.org/software/autoconf
- GNU C/C++: http://gcc.gnu.org
- GNU make: http://www.gnu.org/software/make
- Flex: http://flex.sourceforge.net
- Bison: http://www.gnu.org/software/bison
- GNU Emacs: http://www.gnu.org/software/emacs
- GNU ltdl (part of the libtool software): http://www.gnu.org/software/libtool
- Mercurial: http://mercurial.selenic.com (There's also a very nice Windows frontend, TortoiseHg, see http://tortoisehg.bitbucket.org.)

The GNU multiprecision library or some compatible replacement is required for Pure's bigint support. Instead of GMP it's also possible to use MPIR. You can find these here:

- GMP: http://www.gnu.org/software/gmp
- MPIR: http://www.mpir.org

If you have both GMP and MPIR installed, you can specify `--with-mpir` when configuring Pure to indicate that Pure should be linked against MPIR. Note that using this option might cause issues with some Pure modules which explicitly link against GMP. If you run into any such problems then you should build MPIR with the `--enable-gmpcompat` configure option so that it becomes a drop-in replacement for GMP (in this case the `--with-mpir` option isn't needed when configuring Pure).

In addition, Pure 0.48 and later also require the GNU multiprecision floating point library:

- MPFR: http://www.mpfr.org

(Pure doesn't really have built-in support for MPFR numbers, this is provided through a separate pure-mpfr addon module instead, please check the Pure website for details. However, there is now some support for printing both GMP and MPFR numbers in the `printf` and `scanf` functions of the `system` module, for which the MPFR library is needed.)

To make interactive command line editing work in the interpreter, you'll also need GNU readline or some compatible replacement such as BSD editline/libedit:

- GNU readline: http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html
- BSD editline/libedit: http://www.thrysoee.dk/editline

We recommend GNU readline because it's easier to use and has full UTF-8 support, but in some situations BSD editline/libedit may be preferable for license reasons or because it's what the operating system provides. Pure's configuration script automatically detects the presence of both packages and also lets you disable readline and/or editline support using the `--without-readline` and `--without-editline` options.

Step 2. Get and unpack the latest LLVM sources.

You can find these at http://llvm.org/releases/download.html.

You only need the llvm-2.x or 3.x tarball which contains the LLVM library as well as most of the LLVM toolchain. LLVM 3.1 is the latest stable release at the time of this writing. LLVM versions 2.5 thru 3.1 have all been tested and are known to work with Pure. We really recommend using LLVM 2.8 or later, however, because LLVM has improved considerably in recent releases. (Support for older versions may be dropped in the future.)

The latest LLVM from svn might work as well, but we don't guarantee this. While we're committed to make Pure work with new LLVM versions as they become available, we're not able to track LLVM development in the trunk very closely. So you might run into intermittent compilation problems, bugs and other incompatibilities when going with the svn version.

At this point we also recommend getting an LLVM-capable C/C++ compiler. This is completely optional, but you'll need it to take advantage of the new bitcode loader in Pure 0.44 and later. The easiest way to go is to take the clang-2.x or 3.x tarball which corresponds to your LLVM version, unpack its contents into the LLVM tools directory and rename the `clang-x.y` directory to just `clang`. The clang compiler will then be built and installed along with LLVM. You can also use llvm-gcc or dragonegg instead, please see Installing an LLVM-capable C/C++ Compiler below for further details.

Note: Some (older) Linux and BSD distributions provide LLVM packages and ports which are compiled with wrong configure options and are thus broken. If the Pure interpreter segfaults on startup or fails its test suite (`make check`) then you should check whether there's a newer LLVM package available for your system, or compile LLVM yourself.

Step 3. Configure, build and install LLVM as follows:

```
$ cd llvm-3.1.src
$ ./configure --enable-shared --enable-optimized --enable-targets=host-only
$ make
$ sudo make install
```

LLVM 2.7 and earlier may also require the flags `--disable-assertions --disable-expensive-checks` to disable some features which make LLVM slow and/or buggy on some systems. With LLVM 2.8 and later these options aren't needed any more.

Note that the `--enable-shared` option builds and installs LLVM as a shared library, which is often preferable if you're running different LLVM-based tools and compilers on your system.

This requires LLVM 2.7 or later and may be broken on some systems. You can always leave out that option, in which case LLVM will be linked statically into the Pure runtime library. (You can also force Pure to be linked statically against LLVM even if you have the shared LLVM library installed, by configuring Pure with the `--with-static-llvm` flag. This may be useful if you plan to deploy the Pure runtime library on systems which don't have LLVM installed.)

Note: With LLVM 2.5, on x86-64 systems you have to add `--enable-pic` to the configure command, so that the static LLVM libraries can be linked into the Pure runtime library. (Do *not* add this option when compiling on a 32 bit system, it's broken there.) With LLVM 2.6 and later this option isn't needed anymore. See the comments on 32/64 bit support in the System Notes section below for details.

Also note that the configure flags are for an optimized (non-debug) build and disable all compilation targets but the one for your system. You might wish to play with the configure options, but note that some options (especially `--enable-expensive-checks`) make LLVM very slow and may even break the Pure interpreter on some systems.

Step 4. Get and unpack the Pure sources.

These can be downloaded from http://pure-lang.googlecode.com. The latest source tarballs can always be found under the "Featured Downloads" there.

Step 5. Configure, build and install Pure as follows (`x.y` denotes the current Pure version number):

```
$ cd pure-x.y
$ ./configure --enable-release
$ make
$ sudo make install
```

The `--enable-release` option configures Pure for a release build. This is recommended for maximum performance. If you leave away this option then you'll get a default build which includes debugging information and runtime checks useful for the Pure maintainers, but also runs considerably slower.

To find out about other build options, you can invoke configure as `./configure --help`.

The `sudo make install` command installs the pure program, the runtime.h header file, the runtime library libpure.so, a Pure pkg-config file (pure.pc) and the library scripts in the appropriate subdirectories of /usr/local; the installation prefix can be changed with the `--prefix` configure option, see Other Build And Installation Options for details. (The runtime.h header file is not needed for normal operation, but can be used to write C/C++ extensions modules, if you need to access and manipulate Pure expressions from C/C++.)

In addition, if the presence of GNU Emacs was detected at configure time, then by default pure-mode.el and pure-mode.elc will be installed in the Emacs site-lisp directory. make tries to guess the proper location of the site-lisp directory, but if it guesses wrong or if you want to install in some custom location then you can also set the `elispdir` make variable accordingly. If you prefer, you can also disable the automatic installation of the elisp files

by running configure with `./configure --without-elisp`. (In that case, it's still possible to install the elisp files manually with `make install-el install-elc`.)

On some systems you have to tell the dynamic linker to update its cache so that it finds the Pure runtime library. E.g., on Linux this is done as follows:

```
$ sudo /sbin/ldconfig
```

After the build is complete, you can (and should) also run a few tests to check that Pure is working correctly on your computer:

```
$ make check
```

(This can be done before actually installing Pure, but make sure that you first run `ldconfig` or similar if you installed LLVM as a shared library, otherwise `make check` may fail simply because the LLVM library isn't found.)

If all is well, all tests should pass. If not, the test directory will contain some `*.diff` files containing further information about the failed tests. In that case please zip up the entire test directory and mail it to the author, post it on the Pure mailing list, or enter a bug report at http://code.google.com/p/pure-lang/issues/list. Also please include precise information about your platform (operating system and cpu architecture) and the Pure and LLVM versions and/or source revision numbers you're running.

Note that `make check` executes the run-tests script which is generated at configure time. If necessary, you can also run individual tests by running run-tests directly (e.g., `./run-tests test/test020.pure test/test047.pure`) or rerun only the tests that failed on the previous invocation (`./run-tests -f` or, equivalently, `make recheck`).

Also note that MSYS 1.0.11 (or at least the diffutils package from that version) is required to make `make check` work on Windows. Also, under MS Windows this step is expected to fail on some math tests in test020.pure; this is nothing to worry about, it just indicates that some math routines in Microsoft's C library aren't fully POSIX-compatible. The same applies to BSD systems.

If Pure appears to be broken on your system (`make check` reports a lot of failures), it's often because of a miscompiled LLVM. Please review the instructions under step 3, and check the System Notes section to see whether your platform is known to have issues and which workarounds may be needed. If all that doesn't help then you might be running into LLVM bugs and limitations on not-so-well supported platforms; in that case please also report the results of `make check` as described above, so that we can try to figure out what is going on and whether there's a fix or workaround for the problem.

Note: If you have one of the LLVM C/C++ compilers installed (see Installing an LLVM-capable C/C++ Compiler), you can use those to compile Pure by passing the appropriate compiler names on the configure line:

```
$ ./configure --enable-release CC=clang CXX=clang++
```

Or, when using llvm-gcc:

```
$ ./configure --enable-release CC=llvm-gcc CXX=llvm-g++
```

llvm-gcc 4.2 and clang 2.8 or later should build Pure cleanly and pass all checks.

Step 6. Download and install the online documentation as follows:

```
$ sudo make install-docs
```

This isn't necessary to run the interpreter, but highly recommended, as it gives you a complete set of manuals in html format which covers the Pure language and interpreter, the standard library, and all addon modules available from the Pure website. You can read these manuals with the `help` command in the interpreter. You also need to have a html browser installed to make this work. By default, the interpreter assumes w3m (a text-based browser), you can change this by setting the `BROWSER` or the `PURE_HELP` variable accordingly.

By default, the `install-docs` target requires a working Internet connection and the wget command. Instead, you can also download the pure-docs-x.y.tar.gz tarball manually and then install the documentation from the downloaded tarball (the x.y version number of the documentation tarball should correspond to your interpreter version):

```
$ sudo make install-docs docs=pure-docs-x.y.tar.gz
```

As a bonus, downloading the package manually also gives you the documentation in pdf format, so that you can print it if you like.

Step 7. The Pure interpreter should be ready to go now.

Run Pure interactively as:

```
$ pure
Pure 0.56 (x86_64-unknown-linux-gnu) Copyright (c) 2008-2011 by Albert Graef
(Type 'help' for help, 'help copying' for license information.)
Loaded prelude from /usr/local/lib/pure/prelude.pure.
```

Check that it works:

```
> 6*7;
42
```

Read the online documentation:

```
> help
```

Exit the interpreter (you can also just type the end-of-file character at the beginning of a line, i.e., `Ctrl-D` on Unix):

```
> quit
```

You can also run the interpreter from GNU Emacs (see below), and for Windows there is a nice GUI application named "PurePad" which makes it easy to edit and run your Pure scripts.

35.3 Emacs Pure Mode

This step is optional, but if you're friends with Emacs then you should definitely give Pure mode a try. This is an Emacs programming mode which turns Emacs into an advanced IDE to edit and run Pure programs. If Emacs was detected by configure then after running `make` and `sudo make install` the required elisp files should already be installed in the Emacs site-lisp directory (unless you specifically disabled this with the `--without-elisp` configure option).

Note: make tries to guess the Emacs installation prefix. If it gets this wrong, you can also set the make variable `elispdir` to point to your site-lisp directory. (In fact, you can specify any directory on Emacs' loadpath for `elispdir`.)

Before you can use Pure mode, you still have to add some stuff to your .emacs file to load the mode at startup. A minimal setup looks like this:

```
(require 'pure-mode)
(setq auto-mode-alist
      (cons '("\\.pure\\(rc\\)?$" . pure-mode) auto-mode-alist))
(add-hook 'pure-mode-hook 'turn-on-font-lock)
(add-hook 'pure-eval-mode-hook 'turn-on-font-lock)
```

This loads Pure mode, associates the .pure and .purerc filename extensions with it, and enables syntax highlighting.

Other useful options are described at the beginning of the pure-mode.el file. In particular, we recommend installing emacs-w3m and enabling it as follows in your .emacs file, so that you can read the online documentation in Emacs:

```
(require 'w3m-load)
```

Also, you can enable code folding by adding this to your .emacs:

```
(require 'hideshow)
(add-hook 'pure-mode-hook 'hs-minor-mode)
```

These lines should come before the loading of Pure mode in your .emacs, so that Pure mode can adjust accordingly.

Once Emacs has been configured to load Pure mode, you can just run it with a Pure file to check that it works, e.g.:

```
$ emacs examples/hello.pure
```

The online help about Pure mode can be read with `C-h m`. The Pure documentation can be accessed in Pure mode with `C-c h`.

35.4 TeXmacs Mode

Pure 0.56 has minimal support for running Pure as a session in GNU TeXmacs. This is triggered by the (as yet undocumented) `--texmacs` option of the interpreter. The following little TeXmacs plugin file will do the trick:

```
(plugin-configure pure
  (:require (url-exists-in-path? "pure"))
  (:launch "pure -i --texmacs")
  (:session "Pure"))
```

Place this as pure/progs/init-pure.scm in your ~/.TeXmacs/plugins folder. After restarting TeXmacs you should find Pure in TeXmacs' Insert / Session menu. Note that *both* the `-i` and `--texmacs` options are required in the launch command to make this work (you might also want to add the `-q` option to suppress the signon message of the interpreter).

35.5 Installing an LLVM-capable C/C++ Compiler

As already mentioned above, we suggest that you also install a C/C++ compiler with an LLVM backend. Clang, llvm-gcc as well as the new dragonegg gcc plugin are all fully supported by Pure. Pure can be used without this, but then you'll miss out on the LLVM bitcode loader and C/C++ inlining facilities in Pure 0.44 and later. (However, you can always install clang, llvm-gcc and/or dragonegg at a later time to enable these features.)

35.5.1 clang

With LLVM 2.8 and later, we recommend installing clang, the new LLVM-based C/C++ compiler (http://clang.llvm.org/). It's much easier to build, runs faster and has better diagnostics than gcc. Also, as of Pure 0.55 it is the default for compiling inline C/C++ code, so it's the easiest way to go if you want to use that feature.

If you haven't built clang along with LLVM yet, you can now just drop the contents of the clang-x.y tarball into the `llvm-x.y/tools` directory, renaming the resulting `clang-x.y` directory to just `clang`. Then build and install clang as follows:

```
$ cd llvm-3.1.src/tools/clang
$ make
$ sudo make install
```

35.5.2 llvm-gcc

Note: This section applies to LLVM versions up to 2.9. With LLVM 3.0 or later, llvm-gcc is not supported any more and you should use clang or dragonegg instead.

If available, llvm-gcc can be installed either as an alternative or in addition to clang. The main advantage of llvm-gcc over clang is that it has additional language frontends (Ada and Fortran).

Installing llvm-gcc from source actually isn't all that difficult, if a bit time-consuming. Assuming that you have unpacked both the LLVM and the llvm-gcc sources in the same directory, you can build and install llvm-gcc as follows:

```
$ cd llvm-gcc-4.2-2.9.source
$ mkdir obj
$ cd obj
$ ../configure --program-prefix=llvm- --enable-llvm=$PWD/../../llvm-2.9 --enable-languages=c,c++
$ make
$ sudo make install
```

(You might wish to add `fortran` to `--enable-languages` if you also want to build the Fortran compiler, and, likewise, `ada` for the Ada compiler.)

Be patient, this takes a while.

Having installed llvm-gcc, you can add something like the following lines to your shell startup files, so that Pure uses it for inlined C/C++/Fortran code:

```
export PURE_CC=llvm-gcc
export PURE_CXX=llvm-g++
export PURE_FC=llvm-gfortran
```

35.5.3 dragonegg

If you're running LLVM 3.x, then instead of llvm-gcc you should use "DragonEgg" (http://dragonegg.llvm.org/), the new LLVM backend for gcc >=4.5. This is provided in the form of a plugin which, if you have gcc 4.5 or later, readily plugs into your existing system compiler.

If you already have a suitable gcc version, installing DragonEgg is a piece of cake. First, make sure that you have the mpc and gcc plugin development files installed (packages `mpc-dev` and `gcc-plugin-dev` on Ubuntu). Then, after unpacking the dragonegg source tarball or downloading the svn sources, install dragonegg as follows:

```
$ make
$ sudo cp dragonegg.so `gcc -print-file-name=plugin`
```

Finally, add something like the following lines to your shell startup files, so that Pure uses gcc+dragonegg for all inlined C/C++/Fortran code:

```
export PURE_CC="gcc -fplugin=dragonegg"
export PURE_CXX="g++ -fplugin=dragonegg"
export PURE_FC="gfortran -fplugin=dragonegg"
```

(Please also check the README file included in the dragonegg package for further installation and usage instructions. Also, examples/bitcode/Makefile in the Pure distribution

demonstrates how to use gcc+dragonegg as an external compiler to generate LLVM bitcode from the command line.)

35.6 Installing From Development Sources

The latest development version of Pure is available in its Mercurial (Hg) source code repository. You can browse the repository at:

http://code.google.com/p/pure-lang/source/browse/

(You'll notice that the repository also contains various addon modules. See the pure subdirectory for the latest sources of the Pure interpreter itself.)

Note that if you're going with the development sources, you'll also need fairly recent versions of autoconf, flex and bison (autoconf 2.63, flex 2.5.31 and bison 2.3 should be ok).

To compile from the development sources, replace steps 4 and 5 above with:

Step 4'. Fetch the latest sources from the repository:

```
$ hg clone http://pure-lang.googlecode.com/hg pure-lang
```

This clones the repository and puts it into the pure-lang subdirectory in the current directory. (Project members please use https: instead of http: if you're planning to push any changes back to the server.) This step needs to be done only once; once you've cloned the repository, you can update it to the latest revision at any time by running `hg pull -u`.

Step 5'. Configure, build and install Pure.

This is pretty much the same as with the distribution tarball, except that you need to run 'autoreconf' once to generate the configure script which isn't included in the source repository.

```
$ cd pure-lang/pure
$ autoreconf
$ ./configure --enable-release
$ make
$ sudo make install
```

(Don't forget to also run `make check` to make sure that the interpreter is in good working condition.)

Step 6'. In addition, you can also build and install a recent snapshot of the documentation from the repository.

You need to have a recent Sphinx version installed to do that; you can find this at http://sphinx.pocoo.org/. Have a look at the Makefile in the pure-lang/sphinx subdirectory or type `make help` there for instructions.

Alternatively, a ready-made recent snapshot of the documentation in html and pdf formats is also available in its own repository, which can be cloned as follows:

```
$ hg clone http://docs.pure-lang.googlecode.com/hg pure-lang-docs
```

35.7 Other Build and Installation Options

The Pure configure script takes a few options which enable you to change the installation path and control a number of other build options. Moreover, there are some environment variables which also affect compilation and installation.

Use `./configure --help` to print a summary of the provided options.

35.7.1 Installation Path

By default, the pure program, the runtime.h header file, the runtime library, the pure.pc file and the library scripts are installed in /usr/local/bin, /usr/local/include/pure, /usr/local/lib, /usr/local/lib/pkg-config and /usr/local/lib/pure, respectively. This can be changed by specifying the desired installation prefix with the `--prefix` option, e.g.:

```
$ ./configure --enable-release --prefix=/usr
```

In addition, the `DESTDIR` variable enables package maintainers to install Pure into a special "staging" directory, so that installed files can be packaged more easily. If set at installation time, `DESTDIR` will be used as an additional prefix to all installation paths. For instance, the following command will put all installed files into the tmp-root subdirectory of the current directory:

```
$ make install DESTDIR=tmp-root
```

Note that if you install Pure into a non-standard location, you may have to set `LD_LIBRARY_PATH` or a similar variable so that the dynamic linker finds the Pure runtime library, libpure.so. Also, when compiling and linking addon modules you might have to set `C_INCLUDE_PATH` and `LIBRARY_PATH` (or similar) so that the header and library of the runtime library is found. (This will become unnecessary once all addon modules have been converted to use pkg-config, see below, but this isn't the case right now.) On some systems (notably, BSD) this is even necessary with the default prefix, because /usr/local is not in the default search paths.

As of Pure 0.47, Pure also installs a pkg-config file which may be queried by module Makefiles to determine how to build a module and link it against the Pure runtime library; see Pkg-config Support below. This file will usually be installed into `$(prefix)/lib/pkg-config`. Again, if you use a non-standard installation prefix, you will have to tell pkg-config about the location of the file by adjusting the `PKG_CONFIG_PATH` environment variable accordingly, see pkg-config(1) for details.

35.7.2 Tool Prefix and LLVM Version

On some systems the LLVM toolchain may be located in special directories not on the `PATH`, so that different LLVM installations can coexist on the same system. This is often the case, e.g., if LLVM was installed from a binary package.

To deal with this situation, the configure script distributed with Pure 0.55 and later allows you to specify the directory with the LLVM toolchain using the `--with-tool-prefix` configure option. E.g.:

```
$ ./configure --with-tool-prefix=/usr/lib/llvm-3.1/bin
```

This is also the directory where configure will first look for the `llvm-config` script, so that the proper LLVM version is selected for compilation.

You can also specify the desired LLVM version with the `--with-llvm-version` option. This causes configure to look for the `llvm-config-x.y` script on the `PATH`, where `x.y` is the specified version number. If this option isn't specified, the default is to look for the `llvm-config` script on the `PATH` (this should always work if you installed LLVM from source).

If none of these yield a usable `llvm-config` script, configure will try to locate an `llvm-config-x.y` script by iterating through some recent LLVM releases, preferring the latest version if found. If this fails, too, configure gives up and prints an error message indicating that it couldn't locate a suitable LLVM installation. This is a fatal error, so if you see this then you'll either have to install LLVM yourself, or try to locate a suitable LLVM installation and tell configure about it, using the options explained above.

35.7.3 Versioned Installations

Beginning with version 0.4, Pure fully supports parallel installations of different versions of the interpreter. As of Pure 0.21, to enable this you have to specify `--enable-versioned` when running configure:

```
$ ./configure --enable-release --enable-versioned
```

When this option is enabled, bin/pure, include/pure, lib/pure, lib/pkg-config/pure.pc and man/man1/pure.1 are actually symbolic links to the current version (bin/pure-x.y, include/pure-x.y etc., where x.y is the version number). If you install a new version of the interpreter, the old version remains available as pure-x.y.

Note that versioned and unversioned installations don't mix very well, it's either one or the other. If you already have an unversioned install of Pure, you must first remove it before switching to the versioned scheme.

It *is* possible, however, to have versioned and unversioned installations under different installation prefixes. For instance, having an unversioned install under /usr and several versioned installations under /usr/local is ok.

35.7.4 Separate Build Directory

It is possible to build Pure in a separate directory, in order to keep your source tree tidy and clean, or to build multiple versions of the interpreter with different compilation flags from the same source tree.

To these ends, just cd to the build directory and run configure and make there, e.g. (this assumes that you start from the source directory):

```
$ mkdir BUILD
$ cd BUILD
$ ../configure --enable-release
$ make
```

35.7.5 Compiler and Linker Options

There are a number of environment variables you can set on the configure command line if you need special compiler or linker options:

- `CPPFLAGS`: preprocessor options (`-I`, `-D`, etc.)
- `CXXFLAGS`: C++ compilation options (`-g`, `-O`, etc.)
- `CFLAGS`: C compilation options (`-g`, `-O`, etc.)
- `LDFLAGS`: linker flags (`-s`, `-L`, etc.)
- `LIBS`: additional objects and libraries (`-lfoo`, `bar.o`, etc.)

(The `CFLAGS` variable is only used to build the pure_main.o module which is linked into batch-compiled executables, see "Batch Compilation" in the manual for details.)

For instance, the following configure command changes the default compilation options to `-g` and adds /opt/include and /opt/lib to the include and library search paths, respectively:

```
$ ./configure CPPFLAGS=-I/opt/include CXXFLAGS=-g LDFLAGS=-L/opt/lib
```

More details on the build and installation process and other available targets and options can be found in the Makefile.

35.7.6 Predefined Build Types

For convenience, configure provides some options to set up `CPPFLAGS` and `CXXFLAGS` for various build types. Please note that most of these options assume gcc right now, so if you use another compiler you'll probably have to set up compilation flags manually by using the variables described in the previous section instead.

The default build includes debugging information and additional runtime checks which provide diagnostics useful for maintainers if anything is wrong with the interpreter. It is also noticeably slower than the "release" build. If you want to enjoy maximum performance, you should configure Pure for a release build as follows:

```
$ ./configure --enable-release
```

This disables all runtime checks and debugging information in the interpreter, and uses a higher optimization level (`-O3`), making the interpreter go substantially faster on most systems.

To get smaller executables with either the default or the release build, add `LDFLAGS=-s` to the configure command (gcc only, other compilers may provide a similar flag or a separate command to strip compiled executables and libraries).

You can also do a "debug" build as follows:

```
$ ./configure --enable-debug
```

This is like the default build, but disables all optimizations, so compilation is faster but the compiled interpreter is *much* slower than even the default build. Hence this build is only recommended for debugging purposes.

You can combine all build types with the `--enable-warnings` option to enable compiler warnings (`-Wall`):

```
$ ./configure --enable-release --enable-warnings
```

This option is useful to check the interpreter sources for questionable constructs which might actually be bugs. However, for some older gcc versions it spits out lots of bogus warnings, so it is not enabled by default.

In addition, there is an option to build a "monolithic" interpreter which is linked statically instead of producing a separate runtime library:

```
$ ./configure --enable-release --disable-shared
```

We strongly discourage from using this option, since it drastically increases the size of the executable and thereby the memory footprint of the interpreter if several interpreter processes are running simultaneously. It also makes it impossible to use batch compilation and addon modules which require the runtime library. We only provide this as a workaround for older LLVM versions which cannot be linked into shared libraries on some systems.

In general, the build options can be combined freely with the variables described in the previous section, but note that `--enable-release` and `--enable-debug` will always overwrite the value of `CXXFLAGS`. If this is a problem then it is best to just set up the required flags manually using the variables described in the previous section.

35.7.7 Running Pure From The Source Directory

After your build is done, you should also run `make check` to verify that your Pure interpreter works correctly. This can be done without installing the software. In fact, there's no need to install the interpreter at all if you just want to take it for a test drive, you can simply run it from the source directory, if you set up the following environment variables (this assumes

that you built Pure in the source directory; when using a separate build directory, you'll have to change the paths accordingly):

LD_LIBRARY_PATH=. This is required on Linux systems so that libpure.so is found. Other systems may require an analogous setting, or none at all.

PURELIB=./lib This is required on all systems so that the interpreter finds the prelude and other library scripts.

After that you should be able to run the Pure interpreter from the source directory, by typing `./pure`.

35.7.8 Other Targets

The Makefile supports the usual `clean` and `distclean` targets, and `realclean` will remove all files created by the maintainer, including test logs and C++ source files generated from Flex and Bison grammars. (Only use the latter if you know what you are doing, since it will remove files which require special tools to be regenerated.)

Maintainers can roll distribution tarballs with `make dist` and `make distcheck` (the latter is like `make dist`, but also does a test build and installation to verify that your tarball contains all needed bits and pieces).

Last but not least, if you modify configure.ac for some reason then you can regenerate the configure script and config.h.in with `make config`. This needs autoconf, of course. (The distribution was prepared using autoconf 2.67.)

35.7.9 Pkg-config Support

Pure 0.47 and later install a pkg-config file (pure.pc) which lets addon modules query the installed Pure for the information needed to build and install a module. Besides the usual information provided by pkg-config, such as `--cflags` and `--libs` (which are set up so that the Pure runtime header and library will be found), pure.pc also defines a few additional variables which can be queried with pkg-config's `--variable` option:

- `DLL`: shared library extension for the host platform
- `PIC`: position-independent code flag if required on the host platform
- `shared`: flag used to create shared libraries on the host platform
- `extraflags`: same as `$shared $PIC`

Together with the `libdir` variable, this provides you with the information needed to build and install most Pure modules without much ado. As of Pure 0.55, pure.pc also defines the `tool_prefix` variable which gives the LLVM toolchain prefix specified at configure time, cf. Tool Prefix and LLVM Version.

If you want to use this information, you need to have pkg-config installed, see http://pkg-config.freedesktop.org. This program should be readily available on most Unix-like plat-

forms, and a Windows version is available as well. An example illustrating the use of pkg-config can be found in the examples/hellomod directory in the sources.

35.8 System Notes

Pure is known to work on recent Linux, Mac OSX and BSD versions under x86, x86-64 (AMD/Intel x86, 32 and 64 bit) and ppc (PowerPC), as well as on MS Windows (AMD/Intel x86, 32 bit). There are a few known system-specific quirks and corresponding workarounds which are discussed below.

35.8.1 All Platforms

Compiling the default and release versions using gcc with all warnings turned on (`-Wall`) might give you the warning "dereferencing type-punned pointer will break strict-aliasing rules" at some point in util.cc with some gcc versions. This is harmless and can be ignored.

If your Pure program runs out of stack space, the interpreter will segfault. This is *not* a bug, it happens because runtime stack checks are disabled by default for performance reasons. You can enable stack checks by setting the `PURE_STACK` environment variable accordingly; see the Pure manual for details. The interpreter will then generate orderly "stack fault" exceptions in case of a stack overflow.

35.8.2 32 Bit Systems

With LLVM 2.5 and earlier, the JIT is broken on x86-32 if it is built with `--enable-pic`, so make sure you do *not* use this option when compiling LLVM <=2.5 on 32 bit systems.

Some older LLVM 2.5 packages for Linux are broken on x86-32 for this reason (this has been reported for Ubuntu 9.04 and Fedora Core 10), the symptom being that the Pure interpreter fails a lot of checks and/or segfaults right at startup. In that case you'll have to find a newer, corrected package or build your own LLVM from source instead.

35.8.3 64 Bit Systems

64 bit systems are fully supported by Pure (as far as LLVM supports them).

With LLVM 2.5 and earlier, building the Pure runtime library (libpure) requires that you configure LLVM with `--enable-pic` so that the static LLVM libraries can be linked into the runtime library. With LLVM 2.6 and later, this option isn't needed anymore.

35.8.4 PowerPC

You'll need Pure >= 0.35 and LLVM >= 2.6. Also make sure that you always configure LLVM with `--disable-expensive-checks` and Pure with `--disable-fastcc`. With these settings

Pure should work fine on ppc (tested on ppc32 running Fedora Core 11 and 12), but note that tail call optimization doesn't work on this platform right now because of LLVM limitations.

35.8.5 Linux

Linux is the primary development platform for this software, and the sources should build out of the box on all recent Linux distributions. Packages for various Linux distributions are also available, please check the Pure website for details.

35.8.6 Mac OSX

Pure should build fine on recent OSX versions, and a port by Ryan Schmidt exists in the MacPorts collection, see http://www.macports.org/. If you install straight from the source, make sure that you use a recent LLVM version (LLVM 2.7 or later should work fine on all flavours of Intel Macs). On PowerPC, you might have to build Pure with `--disable-fastcc`, see the PowerPC section above.

Also note that with at least some versions of the Apple gcc compiler, with all warnings turned on you may get the (bogus) warning "control reaches end of non-void function" a couple of times in interpreter.cc. These are due to a bug in older gcc versions (see http://gcc.gnu.org/bugzilla/show_bug.cgi?id=16558), but they are harmless and can be ignored. These warnings should also go away once Apple upgrades its SDK to a newer gcc version.

35.8.7 BSD

FreeBSD now offers a fairly extensive selection of Pure packages in their distribution.

Compilation from source should also work fine on recent NetBSD and FreeBSD versions if you use Pure 0.33 or later. Also make sure that you install a recent port of LLVM which has the `--enable-optimized` flag enabled.

Building Pure requires GNU make, thus you will have to use gmake instead of make. In addition to gmake, you'll need recent versions of the following packages: perl5, flex, bison, gmp, mpfr and readline (or editline). Depending on your system, you might also have to set up some compiler and linker paths. E.g., the following reportedly does the trick on NetBSD:

```
export C_INCLUDE_PATH=/usr/local/include:/usr/pkg/include
export LIBRARY_PATH=/usr/local/lib:/usr/pkg/lib
export LD_LIBRARY_PATH=/usr/pkg/lib:/usr/local/lib
```

35.8.8 MS Windows

Thanks to Jiri Spitz' perseverance, tireless testing and bug reports, the sources compile and run fine on Windows, using the Mingw port of the GNU C++ compiler and the MSYS envi-

ronment from http://www.mingw.org/. Just do the usual `./configure && make && make install`. You'll need LLVM, of course (which builds with Mingw just fine), and a few additional libraries for which headers and precompiled binaries are available from the Pure website (http://pure-lang.googlecode.com).

However, the easiest way is to just go with the Pure MSI package available on the Pure website. This includes all required libraries and some shortcuts to run the Pure interpreter and read online documentation in html help format, as well as "PurePad", an alternative GUI frontend for editing and running Pure scripts on Windows.

After installing the MSI, you might also want to go to the LLVM website and grab the LLVM toolchain for mingw32/x86. It is sufficient to install the "LLVM binaries" package on your system to make the Pure batch compiler work. Just unzip these into some convenient location on your harddrive and set up `PATH` so that it points to the llvm-x.y directory.

Chapter 36

Running Pure on Windows

This document provides some information pertaining to the Windows version of Pure, available from the Pure website in the form of an MSI package. Please note that the Windows version has a custom directory layout which is more in line with standard Windows applications, and will by default be installed in the standard `Program Files` directory on your system.

Normally, most things should be set up properly after you installed the MSI package, but here are a few things that you should know when running the Windows version:

- The Pure interpreter requires the `PURELIB` environment variable to point to the directory containing the prelude and other library modules, available in the `lib` subdirectory of the Pure program directory. Also, the `PATH` environment variable should contain both the Pure program directory and the `lib` subdirectory, so that you can run the interpreter and compiled programs from the command line. Both environment variables are set automatically during installation. To make this work, you have to install the package with administrator rights.
- The package includes a shortcut to run the Pure interpreter in a command window, as well as a shortcut for the online documentation that you're looking at. It also includes the **PurePad** application, a GUI frontend to the Pure interpreter which lets you edit and run Pure scripts on Windows, see *Using PurePad*. After installation you can find these items in the `Pure` submenu of the Program menu.
- Pure scripts can be edited in any text editor. Syntax highlighting and programming modes are provided for Emacs, Vim and various other popular text editors. After installation you can find these in the `etc` subdirectory of the program directory. Please check the files in this directory for installation instructions.
- The interpreter has a few interactive commands (`ls`, `pwd`, etc.) which require Unix-like utilities. To make these work, we recommend installing the CoreUtils package from the GnuWin32 project, and setting your `PATH` accordingly.

Optional Bits and Pieces

The Windows package contains all that's needed to run Pure programs with the interpreter. However, in order to be able to run the Pure batch compiler and to make full use of the Pure/C interface on Windows, you may need to install some third-party programming tools:

- mingw is a full version of the GNU C/C++ compiler for Windows systems. You'll need this in order to create native executables and libraries with the Pure batch compiler. It is also needed for running the **pure-gen** utility included in this package, which can be used to create Pure interfaces to C libraries from the corresponding C headers. And, last but not least you can also use mingw to compile the LLVM tools and the Pure interpreter yourself, if you prefer that.

  Using mingw 4.4 or later is recommended. There's an installer available at the mingw website, see http://www.mingw.org/wiki/Getting_Started for details. You'll want to install both the C/C++ compilers and the MSYS environment. You'll also have to modify the `PATH` environment variable so that it points to the directory containing the mingw binaries, usually `c:\mingw\bin`.

- The batch compiler also requires the LLVM toolchain for mingw32/x86, available from the LLVM download page. In addition, in order to use the C/C++ code inlining feature of the Pure interpreter, you'll need an LLVM-enabled C/C++ compiler such as clang. (That's pretty much the only option on Windows right now; at the time of this writing, the dragonegg plugin for gcc hasn't been ported to Windows yet.)

  This Pure release has been built and tested with LLVM 3.1, so that is the version that you should get. For your convenience, here is the direct download link of the binary package which contains both the LLVM toolchain and the clang compiler:

  http://llvm.org/releases/3.1/clang+llvm-3.1-i386-mingw32-EXPERIMENTAL.tar.bz2

  You should unpack this tarball (using, e.g., 7-Zip) to a directory on your harddisk (say, `c:\llvm`), and modify the `PATH` environment variable so that it points to the `bin` subdirectory of this folder.

- Finally, the Pure program directory needs to be added to the gcc `LIBRARY_PATH` environment variable, so that some Windows-specific addon libraries are found when linking compiled programs. This should be done automatically during installation, but it's a good idea to check the value of `LIBRARY_PATH` after installation and edit it as needed.

Chapter 37

Using PurePad

The following information is available:

- *Running Pure on Windows*: important release notes that you should read first
- Getting Started: a brief overview of PurePad
- Editing Scripts: manage Pure scripts with PurePad
- Running Scripts: run Pure scripts using the Pure interpreter
- Using the Log: how to interact with the Pure interpreter
- Locating Source Lines: quickly find source lines in error messages from the interpreter

Please also check the sidebar for other available documentation, including *The Pure Manual*, the *Pure Library Manual* and information about various bundled modules. (Note that at present only a subset of all available addon modules is included in the Windows distribution.)

37.1 Getting Started

PurePad is a standard Windows application, with the menus, toolbar and status line you are familiar with (see figure below). Like the standard Windows editor from ancient times, PurePad is a single-document application, i.e., there is only one source file open at any time. The main window is divided into two panes. The upper pane is the source pane which usually contains the Pure script you are currently working with. The lower pane is the log pane in which you interact with the interpreter (input expressions to be evaluated, watch the interpreter's output, etc.). Editing operations, as well as the *Font* and *Tab Stops* commands in the *View* menu and the *Print* and *Print Preview* commands in the *File* menu, always apply to the currently selected pane, i.e., the pane which contains the cursor.

To start up the interpreter you use *Script* → *Run* (`F9`) and wait for the interpreter's `>` prompt to appear. This command will run the Pure interpreter with the script currently shown in

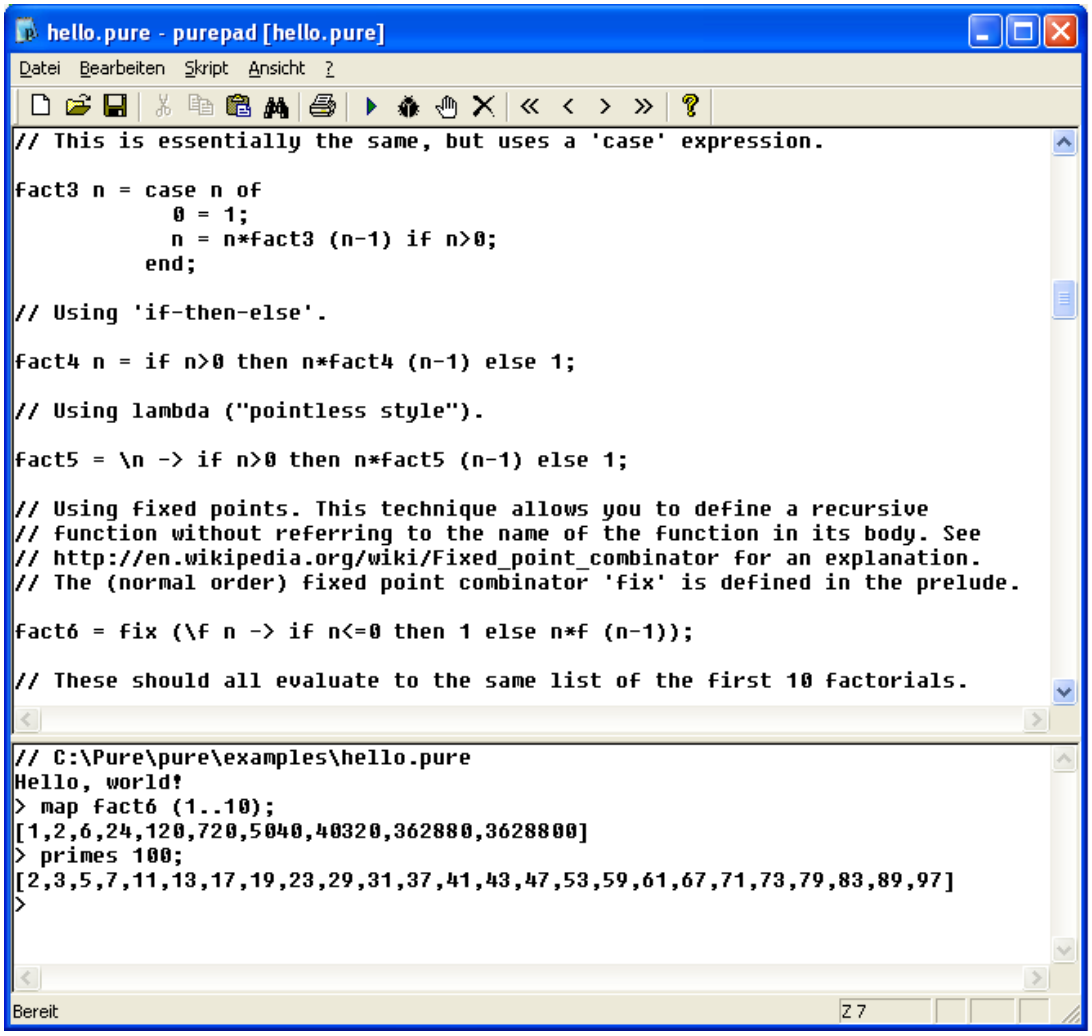

Figure 37.1: PurePad main window.

the source pane (or an empty script if you haven't yet loaded a script using the *File* → *Open* (`Ctrl-O`) command). The cursor will be placed in the log pane (see Using the Log) and you can start typing in expressions and definitions to be evaluated. See Running Scripts for more information on this.

To begin a new script, use the *File* → *New* (`Ctrl-N`) command. Probably the next thing you want to do is to enter your first own little Pure script. For instance, here is a version of the factorial function:

```
fact n = if n>0 then n*fact (n-1) else 1;
```

Enter this equation into the source editor pane, save the script as `fact.pure` with the *File* → *Save* (`Ctrl-S`) command and press `F9`. The cursor will be placed into the log pane and after a while the interpreter's > prompt will appear. You can now type an expression like

```
map fact (1.. 10);
```

and see what happens.

The currently selected pane (source or log) can be printed using the *File* → *Print* (`Ctrl-P`) command; you can also obtain a print preview with the *Print Preview* command in the *File* menu. To change the font and tabulator settings in the current pane, use the *Font* and *Tab Stops* commands in the *View* menu. (Usually it is best to choose a fixed-width font here, like `Fixedsys`, which is also the default. Tab stops are set to 8 by default.)

37.2 Editing Scripts

The *File* menu contains the usual set of operations which let you create new script files, open existing files, save a file that has been edited, preview and print the current file and exit the application. The *View* menu allows you to change tabulator settings (*Tab Stops* option) and the font used for display and printing (*Font* option). Scripts are edited in the upper (source) pane of the main window just as in the Windows Notepad editor. In the *Edit* menu you find the common editing operations (*Undo*, *Cut*, *Copy*, *Paste*, *Select All*, *Find*, *Replace*, and *Go To*, which allows you to jump to the line number you specify). Many of these operations can also be accessed by means of the familiar accelerator keys or the toolbar.

37.3 Running Scripts

Once you have entered your script and saved it in a file, you can run the script using the Pure interpreter. The relevant commands can be found in the *Script* menu:

- *Run* runs your script with the Pure interpreter, in the directory where your source script is located. If the interpreter is already running, it is terminated first.
- *Debug* invokes the interpreter with the built-in debugger. This allows you to trace the calculations ("reductions" in Pure parlance) performed by your script. Note that to

actually debug a function, you must first set a breakpoint using the interpreter's `break` command; please see *The Pure Manual* for details.

- *Break* sends a `Ctrl-C` to the interpreter process. This allows you to interrupt the interpreter when it is doing a time-intensive evaluation, is producing output or is waiting for input.
- *Quit* exits the interpreter. This usually has the same effect as typing `quit` at the interpreter's prompt and is performed automatically when PurePad is exited and the interpreter is still active. It also kills off the interpreter process if it does not terminate within a reasonable amount of time after it has been notified.

As soon as the script has been started, the cursor switches to the log (the lower pane), the interpreter's prompt will appear and you can start typing definitions and expressions, and watch the interpreter print the results. The log pane is an edit control into which you can type text as usual. It also has some special commands which allow you to access an input history and to quickly locate positions in source files, see Using the Log for details.

If you run a script which has errors in it then the Pure compiler will display a list of error messages. To quickly locate the source file positions listed in the error messages, use the commands described in Locating Source Lines.

You can check whether a script is running by taking a look at the title bar of the PurePad window. The name of the previously started script is shown there inside brackets. If it is preceded with the text `Terminated -`, then the script is not currently running, either because the interpreter was exited in a regular fashion, or because some other, unexpected event happened, like a stack overflow.

There are a number of other items in the *Script* menu which deal with the running script and the interpreter's configuration:

- *Open* reopens the previously started script (the one whose name is shown inside brackets in the title bar) in the upper source pane. This operation is useful when you have opened other source files and now want to quickly reload your "main" script.
- The *Prompt* option allows you to change the interpreter's prompt (this only becomes effective when the interpreter is started the next time using the *Run* or the *Debug* command).
- *History File*, *History Size*. With these options you can set the name of the file used to store the input history, and the size of the history, respectively (see Using the Log for details).
- *Reset Log*. When this option is checked, the log is cleared any time a new script is run.

37.4 Using the Log

The log, the lower pane of the PurePad main window, is an edit control which logs both your input to the interpreter and the interpreter's output. In the log you can use all the usual editing operations, as well as the commands in the *Edit* menu. Furthermore, printouts

and print previews of the current contents of the log can be obtained using the corresponding operations of the *File* menu, and the *View* menu allows you to change tabulator settings and the font used for display and printing.

The size (i.e. number of lines) of the log is limited. This limit (500 lines by default, but you can change this manually with the Windows regedit utility) is necessary because each evaluation can produce an arbitrary amount of output, while the text size of a Windows edit control is usually limited to 64 KB.

When the interpreter is currently running, typing `Return` in the last line (the current input line) of the log sends the line (with the prompt removed) to the interpreter. In fact, any line in the log window, not only the last one, can be edited and sent to the interpreter using `Ctrl-Return`. When using this command, the line is copied down to the end of the log.

The log also maintains an input history for the (non-empty) lines you sent to the interpreter. When positioned at the input line, you can browse through this history using the `Ctrl-Up` and `Ctrl-Down` cursor keys (which recall and insert the previous and the next input line, respectively). `Ctrl-PgUp` and `Ctrl-PgDn` go to the first and the last line of the history, respectively. Finally, you can search the history using the `Shift-Ctrl-Up` and `Shift-Ctrl-Down` keys which look for the closest previous (resp. next) history line which matches the text before the current cursor position. The size of the input history (100 lines by default) can be set using the *History Size* option of the *Script* menu. Furthermore, the history is stored in a text file (`PurePadHistory` by default) when PurePad is exited. You can switch to another history file using the *History File* command in the *Script* menu.

37.5 Locating Source Lines

To quickly locate the source line of an error message shown in the log pane, PurePad provides some keyboard commands and toolbar buttons which can be invoked from both the source and the log pane.

- `F3` (*Next Error*, > button) finds and displays the position of the next source line reference in the log, starting below the current line in the log.
- `Shift-F3` (*Previous Error*, < button) finds and displays the position of the previous source line reference in the log, starting above the current line in the log.
- `Ctrl-F3` (*Last Error*, >> button) finds and displays the position of the last source line reference in a sequence of consecutive messages in the log, starting at or above the current line in the log.
- `Shift-Ctrl-F3` (*First Error*, << button) finds and displays the position of the first source line reference in a sequence of consecutive messages in the log, starting at or above the current line in the log.

Each of these commands opens the file indicated by the message (if it is not open already) and sets the cursor to the corresponding line. Furthermore, it marks the message in the log and also displays the source position in the status line. Please note that since PurePad is a single-document application, opening a new source file in this manner closes the file

currently in the source pane. If this file has been modified, you will be prompted to save the file, just as if you used the *File* → *Open* command.

Another useful command in the *View* menu, which is often used in conjunction with the above operations is *To Input Line* (`Esc`) which quickly returns you to the input (i.e., last) line in the log pane.

Module Index

S

t

x

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z