# The Pure Programming Language

Albert Gräf
Department of Computer Music
Johannes Gutenberg University Mainz

February 16, 2010

Copyright © 2009 Albert Gräf. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. See http://www.gnu.org/copyleft/fdl.html.

The latest version of this document and the corresponding LaTeX source can be found at http://pure-lang.googlecode.com/svn/docs/pure-intro.

# Contents

# Preface

This book is about the functional programming language Pure. Pure's distinguishing features are that it is based on term rewriting (a computational model for algebraic expression manipulation), that it provides built-in support for MATLAB-like matrices in addition to the usual list and algebraic data structures, that it uses LLVM (the "Low Level Virtual Machine", see `http://llvm.org`) to compile source programs to fast native code on the fly, and that it makes interfacing to C very easy.

On the surface, Pure looks similar to functional languages of the Miranda and Haskell variety, but under the hood it is a much more dynamic language, offering an interactive interpreter environment and metaprogramming capabilities more akin to Lisp. Pure's algebraic programming style probably appeals most to mathematically inclined programmers who need an advanced tool for solving problems in domains which can be described conveniently in terms of algebraic models. While languages like Haskell and ML already occupy that niche, we think that Pure's feature set is sufficiently different to turn it into a worthwhile alternative. In particular, Pure's interpreter environment and easy extensibility also make it usable as a kind of (compiled) scripting language in a variety of application areas, including system, database, graphics and multimedia programming.

# Acknowledgements

Pure has a small but vivid user community, and I'm very grateful for all the support, discussions, bug reports, patches and other contributions that have helped Pure's development from day one. In particular, I'd like to thank Scott E. Dillard, Rooslan S. Khayrov, Eddie Rucker, Libor Spacek and Jiri Spitz for their substantial code contributions, as well as Toni Graffy and Ryan Schmidt for taking on the arduous tasks of maintaining the SUSE Linux and OSX packages. Thanks are also due to Vili Aapro, Alvaro Castro Castilla, John Cowan, Chris Double, Tim Haynes, John Lunney and Max Wolf for suggesting improvements and pointing out bugs and misfeatures. Last, but not least, a big thank you also goes to the LLVM team for giving us such a powerful sword to fight the complexities of compiler design and implementation.

Chapter 1

# Introduction

Pure is a fairly simple and small language, designed to be easy to learn and use – at least as functional programming languages (FPLs) go. Programs are essentially just 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 nested scopes of local function and variable definitions, anonymous functions (lambdas), exception handling and a built-in macro facility. Macros are defined using rewriting rules, just like ordinary functions. Last but not least, Pure uses the LLVM compiler framework to translate term rewriting systems to native code on the fly, hence programs run reasonably fast and interfacing to C is very easy.

Compared to other modern (as opposed to Lisp-like) functional programming languages, Pure is a bit unusual in that it is essentially a *typeless* language. Keeping with the spirit of term rewriting, all data belongs to the same universe of *terms*, which are formed by applying functions to other terms, with some special notations thrown in for denoting primitive data elements such as numbers and strings. Some terms may evaluate to something else, while others are just normal forms. Pure does *not* distinguish between "constructors" and "defined functions", so that any function symbol can also act as a data constructor. Pure also makes heavy use of *ad-hoc polymorphism*, and in fact it is possible to extend any operation at any time with new defining equations. This makes for a flexible programming model which is much different in style from other, more rigid modern FPLs like Haskell, even though the syntax is superficially similar.

Another fairly unique feature is that Pure, in addition to the usual list and algebraic data structures, also offers built-in matrices (numeric arrays), which is like having Haskell and Octave under one hood, and makes it very easy to express common mathematical operations which go beyond simple arithmetic and operations on scalars. Pure also supports *symbolic matrices* which may contain any kind of Pure value, and *matrix comprehensions* make it easy to program typical algebraic operations. Moreover, Pure's matrix data structure is compatible with the GNU Scientific Library (GSL), so that you can easily interface to GSL's extensive set of numerical algorithms.

Pure normally does eager evaluation, but it also supports lazy data structures through

the notion of lazy *futures* (a.k.a. *thunks*), which was adopted from Alice ML. This makes it possible to work with infinite lists and enables some powerful programming techniques not available in most mainstream languages.

Pure's basic numeric types are (machine) integers, bigints and double precision floating point numbers; the `math.pure` module from the standard library also provides support for complex and rational numbers (including complex integers and complex rationals). In addition, Pure has the same kind of null-terminated strings as C, which are always encoded in UTF-8 internally, so no separate type of Unicode strings is needed. To facilitate interfacing to C, Pure also provides a generic pointer type; there's no lexical representation for these in the language, however, so they have to be created using the appropriate C functions.

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 it 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.

While Pure is already perfectly usable for practical applications and the core language and library are quite stable, there are also some areas which are still under active development. In particular, Pure lacks concurrency and parallelization features right now. An interesting problem there is to design a concurrent programming model that is easy to use, works well in the context of a term rewriting language, and scales well with the number of processors. To these ends, we are currently investigating concurrent futures and automatic parallelization of matrix comprehensions. Another direction for future research is to use available type information to speed up generated code, in particular for numeric operations. Concerning the library, at the time of this writing the GSL interface and scientific programming support are still under development, and we also plan to add object-oriented and dataflow programming extensions in the future.

## 1.1 Getting Pure

The Pure interpreter is free software distributed (mostly) under the GNU Lesser General Public License (LGPL) version 3. Sources and binary packages for various systems are available on the Pure website at `http://pure-lang.googlecode.com`. There you can also find pointers to the Pure mailing list, the wiki and various other bits of documentation which should help you get up and running quickly.

## 1.2 References and Related Work

This document is not a general introduction to functional programming. It will be helpful if you already have at least a passing familiarity with this style of programming, see, e.g., [9], or [19] if you're short on time.

A theoretical introduction to the term rewriting calculus, which Pure is based on, can be found in [1] and [3]. Term rewriting as a programming language was pioneered by Michael O'Donnell [15], and languages based on term rewriting and equational semantics were a fashionable research topic during most of the 1980s and the beginning of the 1990s, two notable examples being the OBJ family of languages [6] and OPAL [4].

As a term rewriting programming language, Pure is most closely related to its predecessor Q [8] and Wouter van Oortmerssen's Aardappel [17], although quite obviously it also heavily borrows ideas from other modern FPLs, in particular Miranda [16], Haskell [10] and Alice ML [14]. Pure's outfix operators were adopted from William Leler's Bertrand language [12], while its matrix support was inspired by MATLAB [13] and GNU Octave [5]. The pattern matching algorithm, which is the main workhorse behind Pure's term rewriting machinery, was invented by the author for his master thesis [7].

Pure also relies on other open source software, most notably the compiler framework LLVM [11] which Pure uses as its backend for doing JIT compilation, as well as the GNU Multiprecision Library (`http://gmplib.org`) for its bigint support.

## 1.3 Etymological Note

People keep asking me what's so pure about "Pure", so let me mention here that the language isn't named that way because it's purely functional (it's not), but because it tries to stay as close to the spirit of the term rewriting calculus as possible without sacrificing some features which I deemed essential for turning Pure into a practical tool for the programmer.

If you prefer, you can also write the name as "PURE" and take this as a recursive acronym for the "Pure Universal Rewriting Engine". This is also slightly misleading, however, because it's not "universal" in the sense that I'd consider Pure the be all and end all of it (I don't, although I tried hard to make it a decent programming language that at least the mathematically inclined might like to use). Rather this is a reference to "universal algebra" [18], the field of mathematics initiated by Alfred North Whitehead and others which gave birth to the notions of equational logic and term rewriting.

## 1.4 Document Roadmap

In the following chapter we start out with an introduction to the Pure interpreter, so that you can follow the examples and try them yourself. Chapter 3 contains a very brief summary of the lexical syntax. Chapters 4 and 5 explain the main elements of Pure

programming, expressions and definitions. Chapter 6 describes in greater detail how term rewriting is implemented in Pure. Chapter 7 discusses advanced topics such as exception handling, lazy evaluation, macro definitions and the C interface. In Chapter 8 you can find more examples illustrating various important programming techniques. The EBNF grammar of Pure can be found in Appendix A.

If you already have an extensive background in functional programming then it's probably enough if you briefly skim through Chapters 4, 5 and 7 before you have a look at the examples in Chapter 8, but we still recommend to carefully read Chapter 6 which describes Pure's basic model of computation by rewriting expressions, which is its most unusual aspect compared to other modern functional programming languages based on the lambda calculus.

## 1.5 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.

Chapter 2

# Using the Interpreter

This chapter assumes that you have already installed the Pure interpreter on your system.[1] Using the interpreter is quite easy. You can invoke it from the command line as follows:

```
$ pure
```

The interpreter then prints its sign-on message and leaves you at its '>' prompt, indicating that it's ready to go. (There are a number of other ways to run Pure if you're not a fan of the command line, see below.)

```
Pure 1.0 (x86_64-unknown-linux-gnu) Copyright (c) 2008, 2009 by Albert Graef
This program is free software, and you are welcome to redistribute it under
certain conditions. There is ABSOLUTELY NO WARRANTY. (Type 'help copying'
for more information.)
Loaded prelude from /usr/local/lib/pure/prelude.pure.

>
```

## 2.1 Interactive Usage

If you invoke the interpreter without arguments (as shown above) or with the `-i` option, it starts up in interactive mode in which you can just begin typing away and evaluate some expressions. Built-in support is available for double precision floating point numbers, integers (both 32 bit machine ints and bigints; the latter are denoted with a trailing `L` symbol), strings, lists and matrices, so you can use the interpreter as a sophisticated kind of desktop calculator right away:

[1] The `INSTALL` file in the distribution explains in great detail how to install Pure from the sources.

```
> 17/12+23;
24.4166666666667
> pow 2 100;
1267650600228229401496703205376L
> "Hello, "+"world!";
"Hello, world!"
> [1,2,3]+[4,5,6];
[1,2,3,4,5,6]
> 1..10;
[1,2,3,4,5,6,7,8,9,10]
> 0:2..10;
[0,2,4,6,8,10]
```

Note that each expression to be evaluated must be terminated with a semicolon.

A convenience for interactive usage is the `ans` function which gives access to the most recent result printed by the interpreter:

```
> 17/12; ans+23;
1.41666666666667
24.4166666666667
```

Alternatively, you can also store intermediate results in variables:

```
> let x = 17/12; x+23;
24.4166666666667
> let x = 1..10; x!5; x!!(3..6);
6
[4,5,6,7]
```

New functions can be defined just as easily. For instance, here's a (rather naive) implementation of the Fibonacci function:[2]

```
> fib n = if n<=1 then n else fib (n-2) + fib (n-1);
> map fib (0..10);
[0,1,1,2,3,5,8,13,21,34,55]
```

At the command prompt, the interpreter also understands a number of special interactive commands. For instance, the `show` command provides a quick means to check what we have accomplished so far:

```
> show
fib n = if n<=1 then n else fib (n-2)+fib (n-1);
let x = [1,2,3,4,5,6,7,8,9,10];
```

It's also possible to just dump our definitions to a file, so that we can edit them in a text editor and reload them with the interpreter later:

[2]Note that there may be a slight delay when your function is executed for the first time. That's nothing to worry about, it's just the JIT compiler kicking in, which compiles the function "just in time" to native code when it is first called.

```
> dump -n fib.pure
> !cat fib.pure
// dump written Sat Apr  4 05:59:05 2009
fib n = if n<=1 then n else fib (n-2)+fib (n-1);
let x = [1,2,3,4,5,6,7,8,9,10];
```

Here we used a shell escape to invoke the system's `cat` command to print the contents of the script file we just saved. This is handy if we want to check what's been written, because the `dump` command itself doesn't provide any feedback.

You can also erase existing definitions:

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

Just `clear` by itself deletes *all* your interactive definitions (after confirmation) so that you can start over:

```
> clear
This will clear all temporary definitions at level #1.
Continue (y/n)? y
> show
>
```

To restore the previous environment, you can reload the `fib.pure` file we saved before:

```
> run fib.pure
> show
fib n = if n<=1 then n else fib (n-2)+fib (n-1);
let x = [1,2,3,4,5,6,7,8,9,10];
```

Remember that `show`, `dump`, `clear`, `run` and the shell escape are special interactive commands with their own syntax; they are not part of the Pure language. In order to facilitate incremental development, the Pure interpreter provides some fairly elaborate means to manipulate definitions interactively. Please refer to the *Pure Manual* for a closer description of these. The manual has a section which explains in much greater detail how the interpreter is used interactively. You can invoke the manual directly from the interpreter as follows:[3]

```
> help
```

Moreover, there's a companion to the Pure Manual, the *Pure Library Manual*, which describes the operations provided in the standard library. The library manual is linked

[3]This requires an external html browser, `w3m` by default; you can set your preferred browser with the `PURE_HELP` or the `BROWSER` environment variable. You can also read the current version of the Pure manual online at `http://pure-lang.googlecode.com/svn/docs/pure.html`, and the Windows package includes the manual as a Windows help file, see Section 2.7.

to in the Pure manual, but you can also quickly look up the description of a function by invoking the `help` command with an argument:

```
> help pow
```

The `help` command has a number of other options which enable you to read different bits of online documentation; try `help pure#online-help` for details.

At this point you might want to explore some of the other modules in the standard library. These can be imported with a **`using`** declaration. E.g., if you want to do some I/O, you'll need the `system.pure` module:

```
> using system;
> puts "Hello, world!";
Hello, world!
14
```

To do more serious math stuff, Pure provides the `math.pure` module:

```
> using math;
> map sin (-pi/2:-pi/4..pi/2);
[-1.0,-0.707106781186547,0.0,0.707106781186547,1.0]
```

Note that this is indeed the C `sin` function being called here. Pure makes it very easy to invoke C functions, you just have to tell the interpreter about the prototype of the function with an **`extern`** declaration.[4] Just for fun, let's compute some random values using the `rand` function from the C library:

```
> extern int rand();
> [rand|i=1..5];
[1025202362,1350490027,783368690,1102520059,2044897763]
```

This also shows a simple example of a *list comprehension*, Pure's workhorse for creating list values through "generator" and "filter" clauses, which mimics traditional set notation in mathematics. Pure also provides *matrix comprehensions* which work analogously but create Octave-style matrices instead. Make sure to check Section 4.8 for a closer description of these. For instance:

```
> eye n = {i==j|i=1..n;j=1..n};
> eye 3;
{1,0,0;0,1,0;0,0,1}
```

Well, we barely scratched the surface here, but I guess that this should be enough to get you started, so that you can begin exploring Pure on your own. To exit the interpreter, just type the `quit` command at the beginning of a line (on Unix systems, typing the end-of-file character `Ctrl-D` will do the same).

```
> quit
```

[4] This always works for functions which are in the C library or the Pure runtime. Functions from other libraries can be called, too, but you must first tell the interpreter about the library with a **`using`** declaration, see Section 7.5 for details.

## 2.2 Debugging

When running interactively, the interpreter also offers a symbolic debugging facility. To make this work, you have to invoke the interpreter with the `-g` option:

```
$ pure -g
```

(This will make your program run *much* slower, so this option should only be used if you actually need the debugger.)

One use of the debugger is "post mortem" debugging. If the most recent evaluation ended with an unhandled exception, you can use the `bt` command to obtain a backtrace of the call chain which caused the exception. 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 debugger can also be used interactively. To these ends you just set breakpoints on the functions you want to debug, using the `break` command. For instance, here is a sample session where we employ the debugger to single-step through an evaluation of the factorial:

```
> 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 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. The prefix ++ denotes reductions which were actually performed during the evaluation and the results that were returned by the function call (printed as "`-->` *x*" where *x* is the return value).

In the above example we just kept hitting the carriage return key to walk through the evaluation step by step. But at the debugger prompt ':' you can also enter various special debugger commands, e.g., to 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. Type the `h` command at the debugger prompt to get a list of the supported commands:

```
: 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   shell escape
? expr  evaluate expression
<cr>    single step (same as 's')
<eof>   step through program, run unattended (same as 'a')
```

More information about the debugger can be found in the corresponding section of the Pure manual.

## 2.3 Running Scripts from the Shell

Pure scripts are just ordinary text files containing Pure code, which can be created with any text editor (special Pure language support is available for some popular text editors, see below). Of course you can run your scripts directly from the command line, as follows:

```
$ pure myscript.pure
```

This executes the script in batch mode, without the interpreter's interactive command loop. (Any number of script files can be specified, which will all be loaded in the indicated order.) Add the `-i` option if you prefer to run the script in interactive mode:

```
$ pure -i myscript.pure
```

This loads and executes the script as above, but then leaves you at the interactive command prompt. You can also add the `-q` option for quiet startup (no sign-on message):

```
$ pure -i -q myscript.pure
```

With the `-x` option it is possible to pass command line parameters to the script (these are available in Pure by means of the predefined `argv` variable, see the manual for details):

```
$ pure -x myscript.pure foo bar baz
```

On Unix systems, the `-x` option also provides a means to run Pure scripts as standalone programs. To these ends, simply make the script executable and add a "shebang" like the following at the beginning of the script:

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

If you have the necessary LLVM tools installed, the interpreter can also create "real" executables which can be run from the shell without a hosting interpreter. Please refer to Section 7.6 for details.

A number of other command line options are available; try `pure -h` for a list of those.

## 2.4 Command Line Editing

When running interactively, the Pure interpreter uses the GNU readline library or some compatible replacement to provide the usual command line editing facilities. Thus the cursor up and down keys can be used to walk through the command history, existing commands can be edited and resubmitted with the carriage return key, etc. The command history is saved in the `.pure_history` file in your home directory between different invocations of the interpreter.

Please note that this feature will only be available if the interpreter was built with readline support. Also, readline support can be disabled at runtime by invoking the interpreter with the `--noediting` option.

## 2.5 Pure and Emacs

If you're friends with GNU Emacs or XEmacs, then this is probably the most convenient way to run Pure for you. To make this work, copy `etc/pure-mode.el` in the sources to your Emacs `site-lisp` directory and add the following lines to your `.emacs` startup file:

```
(require 'pure-mode)
(setq auto-mode-alist (cons '("\\.pure$" . pure-mode) auto-mode-alist))
(add-hook 'pure-mode-hook 'turn-on-font-lock)
(add-hook 'pure-eval-mode-hook 'turn-on-font-lock)
```

More customization options are described at the beginning of the `pure-mode.el` file. Once you have set things up to your liking, you can invoke Emacs with your Pure script as follows:

```
$ emacs myscript.pure
```

Now just type `Ctrl-C Ctrl-C` to run the script. (Use `Ctrl-C Ctrl-D` instead if you want to invoke the interpreter with the `-g` option so that the debugger is available.) This will open a `*pure-eval*` window in Emacs in which you can execute Pure interpreter commands as usual. This has the added benefit that you can get a transcript of your interpreter session simply by saving the contents of the `*pure-eval*` buffer in a file.

The Pure interpreter has an option `--etags` to build an Emacs `TAGS` file, which can be used to quickly locate global declarations and definitions in a Pure script. This option can also be accessed in Emacs Pure mode with the "Make Tags" (`Ctrl-C Alt-T`) command. The Emacs manual has a section which explains how to use tags in the editor.

## 2.6 Other Text Editors

Syntax highlighting support is available for a number of other popular text editors, such as Vim, Gedit and Kate. The Kate support is particularly nice because it also provides

code folding for comments and block structure. See the `etc` directory in the sources. Installation instructions are contained in the language files.

Note that editors like Vim and Kate also provide support for tags, albeit in a different "ctags" format. Tags files in this format can be created using the `--ctags` option of the interpreter, please see the Pure manual for details.

## 2.7 Pure on MS Windows

For MS Windows users we recommend the MSI package provided on the Pure website, which also includes a little Notepad-like frontend to the Pure interpreter. *PurePad*, as it's called, only has a fairly simple text editor which does not support Unicode or syntax highlighting right now, but it does provide all the necessary operations to edit and run your Pure scripts and find the source lines for error messages. It also comes with online help which includes the entire Pure manual. Of course it is also possible to run the Pure interpreter directly from the Windows command line or inside Emacs as described above, but Windows users seem to like *PurePad* because it's easy to use.

Chapter 3

# Lexical Matters

Before we discuss Pure's expression and definition syntax, it is necessary to say a few words about the lexical syntax. This part of the language is pretty conventional, so we won't go into all the boring minutiae here; please see the Pure manual for details.

First and foremost, note that Pure is a *free-format* language, i.e., whitespace is insignificant (unless it is used to delimit other symbols). Thus, in difference to "layout-based" languages like Haskell, you *must* use the proper delimiters (';') and keywords (**end**) to terminate definitions and block structures.

## 3.1 Character Set

Pure fully supports the *Unicode* character set (or, more precisely, ISO 10646). To these ends, scripts can be encoded either as 7 bit ASCII or UTF-8. The latter 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 and constant symbols. The precise rules by which Pure distinguishes "punctuation" (which may only occur in declared operator and constant symbols) and "letters" (identifier constituents) are explained in Appendix A.

## 3.2 Lexical Elements

*Comments* use the same syntax as in C++: `//` for line-oriented, and `/* ... */` for multiline comments. The latter must not be nested.

*Numbers* are the usual sequences of decimal digits, optionally followed by a decimal point and more digits, and/or a scaling factor. In the latter case the sequence denotes a floating point number, such as `1.23e-45`. Simple digit sequences like `1234` denote

integers (32 bit machine integers by default). Using the `0b`, `0x` and `0` prefixes, these may also be written in binary (`0b1011`), hexadecimal (`0x12ab`) or octal (`0177`). The `L` suffix denotes a bigint (`1234L`); other integer constants are promoted to bigints automatically if they fall outside the 32 bit range.

*Strings* are arbitrary character sequences enclosed in double quotes, such as `"abc"` or `"Hello, world!\n"`. 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`). For disambiguating purposes, numeric escapes can also be enclosed in parentheses. E.g., `"\(123)4"` is a string consisting of the character `\123` followed by the digit `4`. Also note that Pure doesn't have a special notation for single characters, these are just strings of length 1 (counting multibyte characters as a single character), such as `"a"` or `"\&copy;"`.

*Identifiers* consist of letters and digits and start with a letter; as usual, the underscore '_' counts as a letter here. Case is significant, so `foo`, `Foo` and `FOO` are all distinct identifiers. The identifier `_` (just an underscore by itself) plays a special role as the *anonymous variable* on the left-hand side of definitions.

*Operators* and *constant symbols* are special symbols which *must* be declared before they can be used, as explained in Section 4.2. Lexically, these can be either ordinary identifiers (like the `and` operator in the standard prelude), or arbitrary sequences of punctuation characters (such as `+` or `~==`). The two kinds of symbols don't mix, so a symbol may either contain just letters and digits, or punctuation, but not both at the same time. In other words, identifiers and punctuation symbols delimit each other, so that you can write something like `x+y` without intervening whitespace, which will be parsed as the three lexemes `x + y`.

Symbols consisting of punctuation are generally parsed using the "longest possible lexeme" a.k.a. "maximal munch" rule. Here, the "longest possible lexeme" refers to the longest prefix of the input such that the sequence of punctuation characters forms a valid (i.e., declared) operator or constant symbol. Thus `x+-y` will be parsed as four tokens `x + - y`, unless you also declare `+-` as an operator, in which cased the same input parses as three tokens `x +- y` instead.

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 and constant symbols.) Moreover, there are some keywords which cannot be used as identifiers:

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

In addition, the interactive commands of the Pure interpreter, like `break`, `clear`, `dump`, `show`, etc., are special when typed at the beginning of the command line, but they can still be used as ordinary identifiers in all other contexts.

Chapter 4

# Expressions

Pure's expression syntax mostly revolves around the notion of *curried function applications* which is ubiquitous in modern functional programming languages. 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. 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. Besides these, Pure provides some special notations for conditional expressions as well as anonymous functions (lambdas) and local scopes of function and variable definitions. The different kinds of expressions understood by the Pure interpreter are summarized in Figure 4.1. We'll describe each of these in the rest of this chapter.

Expressions are parsed according to the following precedence rules: Lambda binds 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 the declared precedence and associativity. Parentheses can be used to group expressions and override default precedences as usual.

## 4.1 Function Applications

The basic means to form compound expressions in Pure is the *function application* which, like in most modern functional languages, is denoted as an invisible infix operation. Thus, $f\ x$ denotes the application of a function $f$ to the argument $x$. Application associates to the left, so $f\ x\ y = (f\ x)\ y$, which is the curried notation of a function $f$ being applied to two arguments $x$ and $y$. More generally, $f\ x_1\ \cdots\ x_n$ denotes the application of $f$ to $n$ arguments $x_1, \ldots, x_n$. This way of writing function applications is named after the American logician Haskell B. Curry who popularized its use through his work on the combinatorial calculus.

Currying makes it possible to derive new functions from existing ones quickly and

| Type | Example | Description |
|---|---|---|
| Lambda | `\x->x+1` | anonymous function |
| Scope | **`case`** `x` **`of`** *rule*`; ...` **`end`**<br>`x` **`when`** *rule*`; ...` **`end`**<br>`x` **`with`** *rule*`; ...` **`end`** | **`case`** expression<br>local variable definition<br>local function definition |
| Conditional | **`if`** `x` **`then`** `y` **`else`** `z` | conditional expression |
| Simple | `x+y`, `-x`, `x mod y`, `not x`<br>`sin x`, `max a b` | operator application<br>function application |
| Primary | `4711`, `1.2e-3`<br>`"Hello, world!\n"`<br>`foo`, `x`, `(+)`<br>`[1,2,3]`, `{1,2;3,4}`, `(1,2,3)`<br>`[x,-y \| x=1..n; y=1..m; x<y]`<br>`{i==j \| i=1..n; j=1..m}` | number<br>string<br>function or variable symbol<br>list, matrix, tuple<br>list comprehension<br>matrix comprehension |

Figure 4.1: Expression types, in order of increasing precedence.

easily by just omitting arguments, which yields an *unsaturated* or *partial application*. Conversely, an application of a function which supplies all needed parameters and is thus "ready to go" is called *saturated*. E.g., taking the prelude function `max` as an example, the partial application `max 0` can be used to denote the function which just returns its argument if it's nonnegative, and zero otherwise. This works because, for any given `y`, `(max 0) y = max 0 y` is just the maximum of `0` and `y`. For instance:

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

Another important point worth mentioning here is that, as a language based on term rewriting, Pure does *not* distinguish between "defined" and "constructor" symbols at all. *Every* function symbol can act as a constructor if it happens to occur in a normal form; we'll discuss this in greater detail in Chapter 6.[1] "Pure" constructors are just a special case of these, namely functions without any defining equations, in which case an application of the function symbol will always be a normal form. For instance, assuming that `foo` and `bar` are function symbols not defined anywhere, you'll get:

```
> foo 77 (bar 99);
foo 77 (bar 99)
```

This makes it possible to represent hierarchical data structures in an algebraic way. No data type declarations are needed for this; you only have to pick a suitable selection

[1]Note that constructors in functional programming are not the same thing as constructors in object-oriented programming. The latter is a special kind of function used to construct an object, whereas the constructors we talk about here are never executed; they are just literal symbols which can be applied to some arguments.

of constructor symbols to distinguish between the different kinds of aggregate structures. For instance, a binary tree data structure might be denoted using terms like `bin 0 (bin 1 nil nil) nil`, employing a ternary function symbol `bin` to represent the interior nodes, and a nonfix (constant) symbol `nil` to denote the leaves of the tree. Section 5.3 shows how to define functions operating on such structures.

## 4.2 Operators

Pure also provides prefix, postfix, outfix and infix notation for special operator symbols, so that customary mathematical operations can be written in a more convenient and familiar way. Operators are just syntactic sugar; by enclosing an operator in parentheses you can always turn it into an ordinary function symbol, such as `(+)` and `(not)`. Thus `2*x` is exactly the same as the curried application `(*) 2 x`, and partial applications work as usual, too; e.g., `(+) 1` is the successor function and `(/) 1` the reciprocal.

An operator symbol may either be an ordinary identifier or consist entirely of punctuation. Multiple symbols in the same operator declaration must be separated with whitespace. All operator symbols must be declared before they are first used, by means of a *fixity* declaration which takes one of the following forms:

- **`infix`** *n symbol* ... **`;`** **`infixl`** *n symbol* ... **`;`** **`infixr`** *n symbol* ... **`;`**
  Declares binary (non-, left- or right-associative) infix operators.
- **`prefix`** *n symbol* ... **`;`** **`postfix`** *n symbol* ... **`;`**
  Declares unary (prefix or postfix) operators.
- **`outfix`** *left-symbol right-symbol* ... **`;`**
  Declares outfix (bracket) symbols.
- **`nonfix`** *symbol* ... **`;`**
  Declares nonfix (constant) symbols.

The precedence level *n* of infix, prefix and postfix symbols must be a nonnegative integer (larger numbers indicate higher precedences, 0 is the lowest level).[2] Alternatively, the precedence level can also be given by an existing operator symbol in parentheses. In either case, the precedence is followed by the list of operator symbols to be declared. At each level, non-associative operators have the lowest precedence, followed by left-associative, right-associative, prefix and postfix symbols. Function application binds stronger than any of these, thus compound arguments in function applications have to be parenthesized, as in `sin (x-1)`.

[2] In theory, the number of precedence levels is unlimited, but for technical reasons the current implementation actually requires that precedences can be encoded as unsigned 24 bit values. This amounts to 16777216 different levels which should be enough for almost any purpose.

Pure also provides Bertrand-style *outfix operators* which are unary operators taking the form of bracket structures. These symbols always come in pairs of matching left and right brackets and have highest precedence. For instance, the following declaration introduces `BEGIN` and `END` as a pair of matching brackets. Syntactically, these are used like ordinary parentheses, but actually they are unary operators which can be defined in your program just like any other operation.

```
outfix BEGIN END;
BEGIN a,b,c END;
```

Like the other kinds of 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)`.

Last but not least, symbols can also be declared as *nonfix*, which turns the given symbols into "operators without operands", i.e., constant symbols. These work pretty much like ordinary identifiers, but are always treated as literal constants, even in contexts where an identifier would otherwise denote a variable (cf. Section 4.3). For instance:

```
nonfix nil;
null nil = 1;
```

An important point worth mentioning here is that the arities of symbols in operator or constant symbol declarations are really just syntax. Those declarations are needed because these symbols are special and are parsed differently (even at the lexical level). You can think of them as an extension of the underlying Pure grammar. Also, on the syntactic level all ordinary function symbols are in fact nullary, and function application is an invisible infix operator.

On the other hand, on the semantic level we're only talking about curried function applications. Here, we may attach an arity to the function represented by a symbol (no matter whether it's an ordinary function symbol or a special operator or constant symbol). This *semantic* arity is the number of arguments we have to supply so that an application of the function becomes saturated (i.e., is ready to be executed), which only depends on the definition of the function (cf. Section 5.3). Note that a symbol might not have a semantic arity at all, namely if it's a *pure constructor* which doesn't have any defining equations.

Thus, when we talk about the arity of a symbol, we may be talking about either its syntactic or its semantic arity; this will usually be clear from the context. In the case of operators, syntactic and semantic arity typically match up, but that's not required. E.g., here is a somewhat contrived example of an infix operator `foo` which actually takes *three* arguments to be saturated:

```
infix 0 foo;
(foo) x y z = x*z+y;
```

Thus `foo` is binary syntactically, but ternary semantically, so if you type `5 foo 8` you get an unsaturated application (i.e., a function) which multiplies its "missing" argument by `5` and then adds `8`:

```
> map (5 foo 8) (1..5);
[13,18,23,28,33]
```

The prelude provides the `arity` and `nargs` functions to determine the syntactic and semantic arity of a symbol or application, respectively.

### 4.2.1 Predefined Operators

The signature of predefined operators declared in the prelude is shown in Figure 4.2. (Note the generous "spacing" of the precedence levels, which makes it easy to sneak in additional operator symbols between existing levels if you have to.)

```
infixl  1000   $$ ;               // sequence operator
infixr  1000   $ ;                // right-associative application
infixr  1100   , ;                // pair (tuple)
infix   1200   => ;               // mapsto constructor
infix   1300   .. ;               // arithmetic sequences
infixr  1400   || ;               // logical or (short-circuit)
infixr  1500   && ;               // logical and (short-circuit)
prefix  1600   ~ ;                // logical negation
infix   1700   < > <= >= == ~= ;  // relations
infix   1700   === ~== ;          // syntactic equality
infixr  1800   : ;                // list cons
infix   1900   +: <: ;            // complex numbers (cf. math.pure)
infixl  2000   << >> ;            // bit shifts
infixl  2100   + - or ;           // addition, bitwise or
infixl  2200   * / div mod and ;  // multiplication, bitwise and
infixl  2200   % ;                // exact division (cf. math.pure)
prefix  2300   not ;              // bitwise not
infixr  2400   ^ ;                // exponentiation
prefix  2500   # ;                // size operator
infixl  2600   ! !! ;             // indexing, slicing
infixr  2700   . ;                // function composition
prefix  2800   ' ;                // quote
postfix 2900   & ;                // thunk
```

Figure 4.2: Operators declared in the prelude, in order of increasing precedence.

At first sight, this signature looks a bit unwieldy, but in fact there are just a few groups of important operations which are described below. The basic arithmetic and logical operators are summarized in the following table:

| | |
|---|---|
| Arithmetic | `+ - * / % ^ div mod` |
| Comparisons | `< > <= >= == ~= === ~==` |
| Logic operations | `~ && ||` |
| Bit operations | `not and or << >>` |

These operations are mostly primitives which are compiled directly to native code if they are applied to the appropriate operands. Here is a brief description of these operators:

- `-x`, `x+y`, `x-y`, `x*y`, `x/y`: These are the usual arithmetic operations which work on all kinds of numbers. Unary minus always has the same precedence as binary minus in Pure. '`/`' is Pure's *inexact division* operation which *always* yields double results. The '`+`' operator also denotes concatenation of strings and lists.

- `x div y`, `x mod y`: Integer division and modulus. These work with both machine ints and bigints in Pure.

- `x%y`: Pure's *exact division* operator. This produces rational numbers and requires the `math` module to work.

- `x^y`: Exponentiation. Like '`/`', this always yields double results. (The prelude also provides the `pow` function to compute exact powers of ints and bigints.)

- `x<y`, `x>y`, `x<=y`, `x>=y`, `x==y`, `x~=y`: Comparison operators. `x~=y` denotes inequality. These work as usual on numbers and strings, equality is also defined on lists and tuples. The result is 1 (`true`) if the comparison holds and 0 (`false`) if it doesn't (`false` and `true` are defined as integer constants in the prelude).

- `x===y`, `x~==y`: Syntactic equality. These work on all Pure expressions. `x===y` yields `true` iff `x` and `y` are syntactically equal, i.e., print out the same in the interpreter.

- `~x`, `x&&y`, `x||y`: Logical operations. These take arbitrary machine integers as arguments (zero denotes `false`, nonzero `true`) and are implemented using short-circuit evaluation (e.g., `0&&y` always yields `0`, without ever evaluating `y`). Note that logical negation is denoted as '`~`' rather than with C's '`!`' (which denotes indexing in Pure, see below).

- `not x`, `x and y`, `x or y`: Bitwise logical operations. These are like '`~`', '`&`' and '`|`' in C, but they also work with bigints in Pure.

- `x<<y`, `x>>y`: Bit shift operations. Like the corresponding C operators, but they also work with bigints in Pure.

Most of the remaining operators are either function combinators, specialized data constructors or operations to deal with lists and other aggregate structures (see Section 4.8):

- `x:y`: This is the *list-consing* operation. `x` becomes the head of the list, `y` its tail. This is a constructor symbol, and hence can be used on the left-hand side of a definition for pattern-matching.

- `x..y`: Constructs *arithmetic sequences* (lists of numbers). `x:y..z` can be used to denote sequences with arbitrary stepsize `y-x`.[3] Infinite sequences can be constructed using an infinite bound (i.e., `inf` or `-inf`). E.g., `1:3..inf` denotes the (lazy) list of all positive odd integers.

- `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. The pair constructor is associative in Pure, so that, in difference to lists, tuples are always "flat". More precisely, `(x,y),z` always reduces to `x,(y,z)` which is the canonical representation of the triple `x,y,z`.

- `#x`: The *size* (number of elements) of the string, list, tuple or matrix `x`. (In addition, `dim x` yields the *dimensions*, i.e., the number of rows and columns of a matrix.)

- `x!y`: The *indexing* operation. The origins of this peculiar notation have been lost in the mist of time, but I believe that it comes from the original edition of the Bird/Wadler book [2]. The prelude defines indexing of strings, lists, tuples and matrices. Note that all indices in Pure are zero-based, thus `x!0` and `x!(#x-1)` denote the first and the last element, respectively. In the case of matrices, the subscript may also be a pair of row and column indices, such as `x!(1,2)`.

- `x!!ys`: Pure also provides *slicing* of all indexed data structures. This operation returns the subsequence (string, list, tuple or matrix) of all `x!y` while `y` runs through the elements of the index collection `ys` (this can be either a list or matrix). 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)`.

- `x.y`: This is the function composition operator, as defined by `(f.g) x = f(g x)`, which is useful if you have to apply a chain of functions to some value. For instance, `max x.min y` is a quick way to denote a function which "clamps" its argument between the bounds `x` and `y`.

- `x$y`: The explicit function application operator. You can use this, e.g., if you need to apply a list of functions to corresponding values in a second list as follows: `zipwith ($) [f,g,h] [x,y,z]`. Also note that, since the `$` operator has low priority and is right-associative, it provides a convenient means to write "cascading" function calls like `foo x $ bar $ y+1` which is the same as `foo x (bar (y+1))`.

- `x+:y`, `x<:y`, `x=>y`: These are all specialized data constructors. `+:` and `<:` are used to represent complex numbers in rectangular and polar notation, respectively. Like

[3] Note that in order 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`.

`%`, these require the `math` module to work. The "hash rocket" `=>` is used, e.g., in the `dict` module to denote key-value associations.

- `x$$y`, `x&`, `'x`: These operators are special forms used to execute expressions in sequence, to create "thunks" a.k.a. "futures" which are evaluated lazily, and to defer the evaluation of an expression. See Section 4.9 for details.

### 4.2.2 Unary Minus

The minus symbol plays a somewhat awkward role in the expression syntax because, following mathematical tradition, it is used both as a prefix and an infix operator. Pure adopts the convention that unary minus is at the same precedence level as binary minus. Thus, assuming the precedences declared in the standard prelude, `-x*y` parses as `-(x*y)`, whereas `-x+y` parses as `(-x)+y`. Also note that `(-)` always denotes the binary minus operator. As a remedy, the special predefined function symbol `neg` is provided as an alias for the unary minus operation, thus `neg x` is the same as `-x`.

Moreover, the parser treats unary minus in front of a numeric constant like `-4711` as a negative number rather than an explicit application of the unary minus operation. But syntactically it is still an operator application and thus has to be parenthesized accordingly; e.g., `sin(-1)` denotes an application of the `sin` function to `-1`, whereas `sin-1` is something entirely different, namely an application of binary minus just like `x-y`.

### 4.2.3 Operator Sections

For convenience, Pure also offers a notation for *operator sections* of the form `(`*x*`+)` (left section) or `(+`*x*`)` (right section), which provide a shorthand for partial applications of an infix operator. The meaning of these constructs is given by `(`*x*`+) `*y*` = `*x*`+`*y* and `(+`*x*`) `*y*` = `*y*`+`*x*. Thus, for instance, `(+1)` denotes the successor and `(1/)` the reciprocal function. Note, however, that the construct `(-`*x*`)` is always interpreted as an instance of unary minus; a function which subtracts *x* from its argument can be written as `(+-`*x*`)`.

## 4.3 Patterns

Pattern matching will be covered in much greater depth in subsequent sections. But since this notion pervades all Pure programming, you'll have to learn about it early on, and so we briefly sketch out the relevant concepts below.

Syntactically, a *pattern* is a simple or primary expression (cf. Figure 4.1, p. 20) on the left-hand side of a *rule*. Note that there are a few kinds of primary expressions which have to be excluded here, since they are in fact special primitives which cannot occur in a normal form expression. Specifically, neither matrices nor list or matrix comprehensions are permitted inside patterns. Also, patterns must not contain any of the special

lambda, **case**, **when**, **with** and conditional expression constructs described in subsequent sections.

Semantically, patterns are template expressions to be matched against a subject expression in order to bind the *variables* in the pattern accordingly. The variable binding constitutes a *local scope* in which some other computation can be performed, such as evaluating the right-hand side of a rule in a function definition or **case** expression.

### 4.3.1 The "Head = Function" Rule

This immediately raises an important question: What exactly *are* the variables in a pattern, and what are the function (or constructor) symbols? In mathematical logic, one usually assumes two disjoint sets of function and variable symbols, which may either be declared explicitly or indicated by special typography, but in a programming language this is rather inconvenient.

Instead, Pure uses the following convention: A variable is just an ordinary identifier in a pattern which occurs at a leaf of the expression tree and is *not* the head symbol of a function application. In other words, the head symbol of a function application is always taken as a literal function symbol, while all other identifiers at leaves in the expression tree are considered as variables. This is also called the *head = function* rule, which is quite convenient because it allows us to get away without explicitly declaring the variables. For instance, taking the pattern `foo (bar x) (y:ys)` as an example, you can see at a glance that `foo` and `bar` as well as the list constructor `(:)` are the function symbols, whereas `x`, `y` and `ys` are the variables.

### 4.3.2 Nonfix Symbols

The above approach is quite intuitive and works reasonably well for most kinds of definitions, except that it doesn't take into account symbolic constants (i.e., constructor symbols with zero arguments). As a remedy, Pure allows you to declare such symbols explicitly with a **nonfix** declaration, see Section 4.2 for details. For instance, the declaration '**nonfix** `nil;`' allows you to have a pattern like '`bin x nil nil`', which might denote, e.g., a binary tree with two empty subtrees. Note that without the **nonfix** declaration, the interpreter would mistake the constant `nil` in that pattern for a variable (which would actually result in an error, because of the linearity requirement discussed below).

Nonfix symbols are "precious" in the sense that, once declared, you cannot use them as variables in patterns any more. Therefore you should use **nonfix** declarations sparingly, and only for reasonably expressive and unique identifiers; *never* use them for generic symbols like `f` or `x`. This isn't a problem with non-identifier symbols, as these can never be variables anyway. (Note that nonfix symbols can also be special symbols consisting of punctuation, which aren't permitted in ordinary identifiers.)

### 4.3.3 The Anonymous Variable

There is one variable symbol which gets special treatment in patterns, the *anonymous variable* '_' which matches any value (independently for all occurrences) and does *not* bind a variable. Thus, e.g., `[_,_]` matches any 2-element list, no matter what the elements are. The anonymous variable is also exempt from the "head = function" rule, i.e., it can be used to match (and ignore) any head symbol in a function application, using a pattern like `_ x y z`.

### 4.3.4 Non-Linear Patterns and Syntactic Equality

Pure allows patterns to be *non-linear*, i.e., they may contain multiple occurrences of a variable. Each occurrence of the same variable other than the anonymous variable must be matched to the same value. For instance, here is how you can define a function `uniq` which removes adjacent duplicates from a list:

```
uniq (x:x:xs) = uniq (x:xs);
uniq (x:xs)   = x:uniq xs;
uniq []       = [];
```

Note the non-linearity in the first rule above which indicates that the first two elements of the actual list argument must be the same for this rule to match. The notion of "sameness" employed here is that of *syntactic equality*, i.e., it is checked that the corresponding subterms have the same structure and content. Syntactic equality is also available as an explicit operation `same` in the prelude, so that the first rule is roughly equivalent to:

```
uniq (x:y:xs) = uniq (x:xs) if same x y;
```

Moreover, the prelude also defines the relational operators === and ~== to check for syntactic equality and inequality, so we could have also written this as:

```
uniq (x:y:xs) = uniq (x:xs) if x === y;
```

Note that all these variations of the definition work no matter whether the list elements have ordinary equality defined on them, e.g.:

```
> a==a; a===a;
a==a
1
> uniq [a,b,b,a];
[a,b,a]
```

It is important to note the differences between syntactic equality and the *semantic equality* predicate implemented by the == and ~= operators. For instance, it makes sense to assert that `0==0.0`, even though the integer `0` and the floating point number `0.0` are *not* the same syntactically:

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

In general, it is impossible to predict how semantic equality should be defined for some data structure, and therefore it is only available for data on which it is defined explicitly. The prelude does this for numbers, strings, lists, tuples, matrices and pointers, but leaves it up to the programmer to supply suitable definitions of equality for his own data structures. (This is actually an advantage, since it gives the programmer the freedom to define semantic equality in any way that he sees fit, and it allows the == and ~= operations to be treated in a manner consistent with the other comparison operators such as <= and >.) In contrast, two expressions can *always* be compared for syntactic equality using either a non-linear pattern, the `same` predicate or the === and ~== operators.

### 4.3.5 Type Tags

Another issue which needs special consideration is matching the built-in types of objects (numbers, strings, matrices and pointers). The problem is that you can't write out all "constructors" for the built-in types, as there are infinitely many (or none, as in the case of matrices and pointers which are both constructed using special primitives). Therefore Pure allows you to follow a variable in a pattern (including the anonymous variable) by the symbol '`::`' and one of the special *type tags* `int`, `bigint`, `double`, `string`, `matrix` and `pointer`, to indicate that it can only match a value of the corresponding built-in type.

For instance, here is a "monomorphic" version of the Fibonacci function from Chapter 2 which is restricted to (machine) integer arguments:

```
> fib n::int = if n<=1 then n else fib (n-2) + fib (n-1);
> fib 10; fib 10.0;
55
fib 10.0
```

Type tags can also be applied to user-defined data structures. If the type tag is not any of the built-in type symbols listed above, it is assumed to denote a unary constructor symbol *h* for a given data type, which is then expanded to a special "as" pattern matching values of the form *h y*, see below.

### 4.3.6 "As" Patterns

Pure supports Haskell-style "as" patterns of the form *variable*`@`*pattern* which 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 a definition. Syntactically, "as" patterns are primary expressions; if the subpattern is not a primary expression, it must be parenthesized. For instance, the following function duplicates the head element of a list:

```
> dup xs@(x:_) = x:xs;
> dup [a,b,c];
[a,a,b,c]
```

Here is a little trick which comes in handy if you need to formulate a generic pattern for a function application which binds a variable symbol to the function part of the application (which cannot be done directly because of the "head = function" rule). Since the anonymous variable can match any subterm anywhere, we can use an *anonymous "as" pattern* like `f@_` in order to "escape" the variable symbol `f`. E.g., the following little example demonstrates how you can convert a function application to a list containing the function and all its arguments:

```
> foo x = ac [] x;
> ac xs (x@_ y) = ac (y:xs) x; ac xs x = x:xs;
> foo (a b c d);
[a,b,c,d]
```

Another convenient shorthand is the type tag notation already introduced in the preceding subsection. If *h* is any existing function symbol (more precisely, an identifier) other than the built-in type symbols listed in the preceding subsection, then the notation *x*`::`*h* can be used as an abbreviation for the "as" pattern *x*`@(`*h* `_)`. This enables you to represent a custom data type by designating a special constructor symbol which takes the actual data as its single argument.

Note that this is merely a convention, but it works reasonably well and makes up for the fact that Pure doesn't support data types at the language level. For instance, we might represent points in the plane using a constructor symbol `Point` which gets applied to pairs of coordinates. We equip this data type with an operation `point` to construct a point from its coordinates, and two operations `xcoord` and `ycoord` to retrieve the coordinates:

```
point x y = Point (x,y);
xcoord (Point (x,y)) = x;
ycoord (Point (x,y)) = y;
```

Now 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 = point (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, without knowing anything about the internal representation. We can use these definitions as follows:

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

Some data types in Pure's standard library (specifically, the container data types) are actually represented in this fashion, see the Pure Library Manual for details.

## 4.4 Lambdas

Sometimes you have to deal with more general cases of partial applications which cannot be handled by just omitting parameters like in the `max 0` example in Section 4.1. In such situations *lambda abstractions* a.k.a. anonymous functions come in handy. The customary mathematical notation for these is $\lambda x.y$, where $x$ is the lambda variable and $y$ the body of the lambda function. This denotes a function taking a single parameter $x$ and returning the value of $y$ after substituting all occurrences of $x$ in $y$. Since $\lambda$ is not an ASCII symbol and '`.`' is used for other purposes (cf. Figure 4.2), Pure adopts the Haskell notation, which employs the backslash as an ASCII facsimile of $\lambda$ and an arrow to separate the lambda parameters from the body. E.g., `\x->1/x` is another way to write the reciprocal, while `\x->x/2` halves its argument.

Also like in Haskell, the notation is extended to allow for multiple arguments and argument patterns (cf. Section 4.3). Thus, e.g., `\x y->x*y+1` denotes a lambda taking two arguments `x` and `y`, and `\(x,y)->x*y` yields an anonymous function accepting a pair `x,y` as its (single) argument. (Like the other variable-binding constructs discussed below, a pattern-matching lambda throws an exception if the actual arguments do not match the argument patterns.)

Lambdas bind very weakly in the expression syntax, so they always have to be parenthesized when used in the context of a larger expression. E.g., `(\x->1/x) (2*2)` applies our reciprocal from above to the result of evaluating the argument expression `2*2`, which gives the result `0.25`.

## 4.5 Conditional and Case Expressions

Conditional expressions take the form **`if`** *x* **`then`** *y* **`else`** *z*. Depending on the value of *x*, either *y* or *z* is evaluated and gives the value of the entire expression. For instance, the `max` function is defined in the prelude as follows:

```
max x y = if x>=y then x else y;
```

Multiway conditionals can be realized with the usual **`if`**-**`then`**-**`else`**-**`if`** chains, e.g.:

```
sgn x = if x>0 then 1 else if x<0 then -1 else 0;
```

The condition must yield a truth value, which is just a machine integer in Pure, with zero denoting "false" and any nonzero value denoting "true". Truth values are returned, in particular, by the relational and logic operations (see Figure 4.2, p. 23), but any kind of operation which returns a machine integer does just as well, e.g.:

```
> odd x = x mod 2;
```

```
> if odd 5 then "odd" else "even";
"odd"
```

Reasonably enough, an exception is thrown if the condition fails to evaluate to a truth value (Section 7.2 describes how to deal with such exceptions in a Pure program):

```
> if bogus then 1 else 2;
<stdin>, line 23: unhandled exception 'failed_cond' while evaluating
'if bogus then 1 else 2'
```

The **case** expression provides a dedicated multiway conditional which can also do pattern matching to deconstruct the target value. For instance, here is another definition of the function uniq from Section 4.3.4 which removes adjacent duplicates from a list:

```
uniq xs = case xs of
            x:x:xs = uniq (x:xs);
            x:xs   = x:uniq xs;
            _      = xs;
          end;
```

The individual rules in a **case** expression are considered in the order in which they are written, and each rule constitutes a local scope which binds the variables in the left-hand side pattern of the rule to the corresponding subterms of the subject term, in this case uniq's argument xs. Thus, the first rule handles the case of a list x:x:xs where the first two elements are the same. (Note that the left-hand-side pattern x:x:xs here also rebinds xs locally to the remainder of the list.) In this case, one of the identical elements is removed and uniq is applied recursively to the resulting smaller list x:xs. Otherwise, the next rule is considered which recurses into the list of remaining elements xs in case of a nonempty list x:xs. If that fails as well, the final rule is invoked which has just the anonymous variable _ on the left-hand side and no guard either. Thus this rule always matches and returns the xs argument of uniq unchanged.

In this example, we provided a "catch all" rule, so that the **case** expression never fails. However, we also might have written the definition as follows, in order to ensure that the argument really is a proper list value:

```
uniq xs = case xs of
            x:x:xs = uniq (x:xs);
            x:xs   = x:uniq xs;
            []     = [];
          end;
```

If we now invoke uniq with a non-list value then the modified **case** expression throws a failed_match exception:

```
> uniq 99;
<stdin>, line 38: unhandled exception 'failed_match' while evaluating
'uniq 99'
```

This might be preferable if you want to catch "type errors" as soon as possible. With our previous definition, `uniq 99` returns just `99`.

## 4.6 Local Definitions

Another way to bind local variables in a Pure expression is the **`when`** expression:

$$x \texttt{ when } u_1 = v_1 \texttt{; } u_2 = v_2 \texttt{; ... end}$$

This matches each pattern $u_i$ against the value of each right-hand side $v_i$, binding the variables in the patterns accordingly, and finally evaluates $x$ in the context of these bindings. The bindings are executed in the given order; each $u_i = v_i$ introduces a new nested scope, and each $v_i$ may use all variables introduced in all earlier clauses. A `failed_match` exception is thrown if (and as soon as) a match fails. Note that the simplest case with just a single binding clause, $x$ **`when`** $u = v$ **`end`**, is in fact equivalent to the **`case`** expression **`case`** $v$ **`of`** $u = x$ **`end`**, but the former is usually more convenient. For instance, here is a little definition which lets us compute Fibonacci numbers in pairs (which is much more efficient than the naive definition given in Chapter 2):

```
> fibs n = 0,1 if n<=0;
>        = b,a+b when a,b = fibs (n-1) end;
> fibs 10;
55,89
```

Note the **`when`** expression on the right-hand side of the second equation which is used to extract the results from a recursive invocation of `fibs`.

A similar construct is used to define local functions:

$$x \texttt{ with } f\ x_1\ \cdots\ x_n = y \texttt{; ... end}$$

In difference to **`when`** expressions, this introduces only one local scope which defines all given functions simultaneously. (The syntax of the function definitions is actually more elaborate than what is shown here, you can have any number of local function definitions following the syntactic rules described in Chapter 5.) For instance, the **`with`** expression lets us wrap up our previous definition of `fibs` as a local function in the following alternative definition of the Fibonacci function:

```
> fib n = a when a,b = fibs n end with
>   fibs n = 0,1 if n<=0;
>          = b,a+b when a,b = fibs (n-1) end;
> end;
> map fib (0..10);
[0,1,1,2,3,5,8,13,21,34,55]
```

This kind of "wrapper-worker" design is pretty common in functional programs. A step-by-step explanation of this example and some other variations of the `fib` function can be found in Section 8.1.

The constructs discussed above are similar to Haskell's **`where`** clauses, but note that in Pure they are part of the expression rather than the definition syntax and hence they usually apply to the corresponding subexpression rather than the definition as a whole. (However, like in Haskell it is also possible to have local definitions spanning both the right-hand side and the guard of a rule, see Section 5.2 for details.) Also, there are two different constructs for defining variables and functions. 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 definition binding the local variable `x` by matching the constructor pattern `foo x` against the value `y`. The **`with`** construct does the former, **`when`** the latter.

It is also worth noting here that the variable-binding forms, i.e., lambdas as well as **`case`** and **`when`** expressions, are in fact all reducible to the **`with`** construct, using the following equivalences:

| | | |
|---|---|---|
| `\`$x_1 \cdots x_n$ `->` $y$ | $\equiv$ | $f$ **`with`** $f\ x_1 \cdots x_n$ `=` $y$`;` $f$ `_` $\cdots$ `_ =` $\bot$ **`end`** |
| **`case`** $x$ **`of`** $y_1$ `=` $z_1$`; ...;` $y_n$ `=` $z_n$ **`end`** | $\equiv$ | $f\ x$ **`with`** $f\ y_1$ `=` $z_1$`; ...;` $f\ y_n$ `=` $z_n$`;` $f$ `_ =` $\bot$ **`end`** |
| $x$ **`when`** $y_1$ `=` $z_1$`; ...;` $y_n$ `=` $z_n$ **`end`** | $\equiv$ | $x$ **`when`** $y_n$ `=` $z_n$ **`end`** $\cdots$ **`when`** $y_1$ `=` $z_1$ **`end`** |
| $x$ **`when`** $y$ `=` $z$ **`end`** | $\equiv$ | `(\`$y$ `->` $x$`)` $z$ |

In the above rules, $f$ always denotes a new, nameless function symbol not occurring free in any of the involved subexpressions, and $\bot$ stands for an exception raised in case of a failed match, cf. Section 7.2.

## 4.7 Lexical Scoping

A few remarks about Pure's scoping rules are in order here. The **`when`** and **`with`** expressions, as well as lambda and **`case`** introduce a hierarchy of *local scopes* of identifiers, pretty much like local function and variable definitions in Algol-like block-structured languages. The only unusual thing here is that Pure permits local scopes right in the middle of an expression, instead of only at the definition or statement level.

It is always the *innermost* binding of an identifier which is in effect at each point in the program source. This is determined statically, by just looking at the program text, which is why this scheme is known as *static* or *lexical binding* in the programming literature. For instance:

```
> (x when x = x+1; x = 2*x end) + x;
2*(x+1)+x
```

To understand this result, note that the `x` on the right-hand side of the first local binding, `x = x+1`, refers to a global symbol `x` here (as does the instance of `x` outside of

the **when** expression), which we assume to be unbound in this example.[4] Also note that the above **when** expression is actually equivalent to two nested scopes:

```
> (x when x = 2*x end when x = x+1 end) + x;
2*(x+1)+x
```

Since **when** and **with** are tacked on to the end of the expression they apply to, it is often easier to read these constructs from right to left, i.e., in this example first `x` is bound to `x+1` (employing the global `x`), then it is rebound to `2*x` using the previous value of `x` (which is `x+1` at this point, giving `2*(x+1)`), and finally the same global `x` is added to that.

It goes without saying that this kind of nested expressions can be a little confusing at times, so it's often better to not reuse identifiers too much. E.g., the above expression becomes much clearer if we rewrite it as:

```
> (z when y = x+1; z = 2*y end)+x;
2*(x+1)+x
```

On the other hand, the inflation of symbols which goes with this coding style can also make your code less readable in some cases, so as usual it is up to you to find an approach which matches your own programming style and maintains readability.

Local functions are handled in an analogous fashion to what we have seen above, but there is another subtle issue here, namely that a local function may also refer to other local functions and variables in its own context. This isn't much of a problem as long as the function is only invoked in its original context, but since functions are first-class citizens in Pure, they may easily escape their "home" environment, e.g., if another function returns a local function as its result. In this case, lexical scoping dictates that the local function carries with it the bindings of all local entities it refers to. Such a combination of a local function and its lexical environment is also called a *lexical closure*. For instance, consider the following example of a function `adder` which takes some value `x` as its argument and returns another function `add` which adds `x` to its own argument:

```
> adder x = add with add y = x+y end;
> let a = adder 5; a;
add
> a 5, a 7;
10,12
```

Because of lexical scoping, this works the same no matter what other global or local bindings of `x` are in effect when our instance of `adder` is invoked:

```
> let x = 77; a 5, a 7 when x = 99 end;
10,12
```

[4] As a term rewriting language, Pure has no trouble dealing with unbound symbols, they just stand for themselves.

## 4.8 Primary Expressions

Let us finally consider the primary expressions, which are the basic building blocks of Pure expressions. Some of these coincide with the lexical entities of identifiers and special symbols, strings, integer and floating point numbers already discussed in Chapter 3. In addition, there are some kinds of *compound* primary expressions, namely lists, tuples, matrices and records, as well as list and matrix comprehensions. We describe each of these in turn.

### 4.8.1 Symbols

As already mentioned, symbols come in three different kinds: *Identifiers* like `foo` and `foo_bar77`, depending on the context, may be head symbols in function and constructor applications, or they may denote variables and constant values. *Nonfix* symbols are symbols which denote literal constants; these must be declared and cannot be used as variable symbols (cf. Section 4.3.2). *Operators* (`+`, `-`, `*`, `/`, `div`, `mod`, etc.) are written using infix, prefix, postfix or outfix notation, as the declaration of the operator demands, but are just syntactic sugar for function applications (cf. Section 4.2). By enclosing an operator symbol in parentheses, such as `(+)` or `(mod)`, you turn it into an ordinary function symbol, which is a legal expression just like an identifier or a nonfix symbol. Nonfix and operator symbols can either be identifiers or consist of special punctuation. These are the only symbols that have to be declared explicitly; cf. Section 4.2. Ordinary identifiers are declared implicitly when they are first used.

### 4.8.2 Numbers

Pure has built-in support for three kinds of numeric values, machine integers (32 bit), bigints (arbitrary precision integers, implemented using the GMP a.k.a. GNU Multi-precision Library) and floating point numbers (64 bit). These are all signed quantities. Machine integers use two's complement for representing negative values, while bigints use a sign-magnitude representation (nevertheless, the bitwise arithmetic works on bigints as if they also used two's complement). Pure uses whatever flavour of double precision floating point values are provided by your machine, which usually are IEEE 754 floating point numbers nowadays. Machine int and floating point operations like arithmetic and bit fiddling are all translated to native machine code operations, while bigint operations are implemented using the GMP library.

To distinguish machine integers and bigints, Pure uses the `L` suffix to indicate the latter (`1234L`). Both kinds of integers can be denoted in decimal (`1000`), hexadecimal (`0x3e8`), octal (`01750`) and binary (`0b1111101000`), see Chapter 3 for details. Negative numbers are denoted by prefixing the number with a '`-`' sign (`-1234`, `-1.234e-5`). Syntactically, however, these are actually applications of the unary minus operator and must be parenthesized accordingly, see Section 4.2.

### 4.8.3 Strings

For performance reasons, strings are really a separate data type in Pure, not just lists of characters, although you can easily convert between the two using the `list` and `string` operations defined in the prelude. Also note that single characters are just strings of length 1 in Pure. (Internally, a character may actually consist of multiple bytes, encoding extended Unicode symbols in UTF-8. This provides backward compatibility with ASCII while giving you access to the full Unicode character set.)

The basic list operations (determining the size, indexing, slicing, see Section 4.8.4 below) all work completely analogously on strings:

```
> #"abcd"; "abcd"!1; "abcd"!!(1..2);
4
"b"
"bc"
> "abc"+"xyz";
"abcxyz"
```

Most other list operations are readily supported on strings as well, so that strings can mostly be used as if they were lists of characters (including their use in generator clauses of list comprehensions, see Section 4.8.8). Moreover, the usual arithmetic operations on individual characters are provided, too, which enables you to enumerate sequences of characters. For instance:

```
> upper c = if "a"<=c && c<="z" then c-32 else c;
> string (map upper "The brown fox jumps quickly over the lazy dog.");
"THE BROWN FOX JUMPS QUICKLY OVER THE LAZY DOG."
> string [c+1 | c = "hal"];
"ibm"
> "a".."k";
["a","b","c","d","e","f","g","h","i","j","k"]
```

In addition, strings also support lexicographic comparisons, and it's possible to locate a substring in a string:

```
> "aba"<"abba";
1
> index "abba" "bb";
1
```

There are a number of other operations which you can find in the `strings` module. Together with the indexing and slicing operations these provide a fairly complete set of basic string manipulation functions. (The `system` module also has the usual filename globbing and regular expression routines; we'll have a look at those in Chapter 8.)

### 4.8.4 Lists

*Lists* use the customary bracket notation, as in `[a,b,c]` which is in fact just syntactic sugar for `a:b:c:[]`, where ':' is the list constructor ("cons"), which associates to the right, and `[]` is the empty list. Since ':' is a constructor symbol, it can be used in patterns to deconstruct list values. E.g., the prelude defines the functions `head` and `tail` to extract the head element and the rest of a list ("car" and "cdr" in Lisp parlance) as follows:

```
head (x:xs) = x;
tail (x:xs) = xs;
```

The prelude also defines operations for determining the size of a list, indexing, slicing and list concatenation (recall that list indices are always zero-based):

```
> #[a,b,c,d]; [a,b,c,d]!1; [a,b,c,d]!!(1..2);
4
b
[b,c]
> [a,b,c]+[x,y,z];
[a,b,c,x,y,z]
```

Lists can be compared for equality, by recursively comparing their elements:

```
> [1,2]==[1,2,0];
0
> [1,2,1]~=[1,2,0];
1
```

The prelude also provides the '`..`' operation to create arithmetic sequences:

```
> 1..10;
[1,2,3,4,5,6,7,8,9,10]
> 1:3..10;
[1,3,5,7,9]
> 10:9..0;
[10,9,8,7,6,5,4,3,2,1,0]
```

In addition, the prelude defines a fairly complete set of customary list functions like `head`, `tail`, `init`, `last`, `drop`, `take`, `filter`, `map`, `foldl`, `foldr`, `scanl`, `scanr`, `zip`, `unzip`, etc., which are similar to the list processing facilities provided in ML and Haskell. (In Pure, these also work on strings and matrices accordingly, although these data structures are internally represented as different kinds of C arrays for efficiency and easy interfacing to C.) For instance:

```
> sum = foldl (+) 0; prod = foldl (*) 1;
> sum (1..10); prod (1..10);
55
3628800
```

```
> zipwith (*) [1,2,3] [2,3,2];
[2,6,6]
```

A special feature of lists is that most of the prelude operations can also be used with *lazy* lists a.k.a. *streams*, so that it's possible, e.g., to work with infinite series (this is discussed in detail in Section 7.3):

```
> let s = scanl (+) 0 [1/3^n|n=1..inf];
> s![1,10,20,30]; // this converges to 0.5 fairly quickly
[0.333333333333333,0.499991532456096,0.499999999856601,0.499999999999997]
> s!100;
0.5
```

### 4.8.5 Tuples

*Tuples* are a kind of "flat" list data structure which is commonly used to pass simple aggregate values to functions or return them as results. They are constructed using the right-associative pair constructor `','` and the empty tuple `()`, which work pretty much like `':'` and `[]` (minus the lazy evaluation features that lists provide), but have the following additional properties:[5]

- 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.

Note that these properties imply that tuples can't be nested (if you need this then you should use lists instead). On the other hand, this means that with just the `','` operator you can do *all* basic tuple manipulations (prepend and append elements, concatenate tuples, and do pattern matching):

```
> a,(b,c,d); (a,b,c),d;
a,b,c,d
a,b,c,d
> (a,b,c),(x,y,z);
a,b,c,x,y,z
> foo (2,3) with foo (x,y) = 2*x+y end;
7
```

Tuples thus provide a convenient way to represent plain sequences which don't need an elaborate, hierarchical structure.

[5] Mathematicians should note here that this is not the "official" set-theoretic definition of a tuple, which excludes associativity in order to allow nested tuples. However, many mathematicians effectively use tuples in a way which identifies $X^n \times X^m$ with $X^{n+m}$ because it's just more convenient that way, and this is exactly what Pure provides.

Also note that the parentheses are not really part of the tuple syntax in Pure. Hence `(1,2,3)` prints as just `1,2,3`. However, the parentheses may be needed to override default precedences, e.g., if a tuple value is used as a parameter. There's another case where the parentheses are mandatory, namely if you want to include a tuple in a list or matrix. E.g., `[(1,2),3,(4,5)]` is a three element list consisting of the tuple `1,2`, the integer `3`, and another tuple `4,5`. Likewise, `{(1,2,3)}` is a matrix with a single element, the tuple `1,2,3`.

The prelude defines a few basic list-like operations on tuples, such as determining the size of a tuple with the `#` prefix operator, indexing (`x!i`), slicing (`x!!is`) and comparing for equality (`x==y`, `x~=y`). Moreover, you can easily convert between tuples and lists using the `list` and `tuple` operations.

### 4.8.6 Matrices

Pure also offers matrices, a kind of arrays, as a built-in data structure which provides efficient storage and element access and is tailored for numeric and algebraic applications. These work more or less like their MATLAB and Octave equivalents, but using curly braces instead of brackets. Commas are used to separate the columns of a matrix, semicolons for its rows. For instance:

- `{1,2,3}` is a row vector consisting of machine integers;
- `{1.0;2.0;3.0}` is a column vector of double values;
- `{1,2;3,4}` is a $2 \times 2$ matrix of machine integers;
- `{1L,y+1;foo,bar}` is a "symbolic" matrix (see below).

In fact, the `{...}` construct is rather general, allowing you to construct new matrices from individual elements and/or submatrices, provided that all dimensions match up. E.g., `{{1;3},{2;4}}` is another way to write the $2 \times 2$ matrix `{1,2;3,4}` in "column-major" form. (Internally, however, matrices are always stored in row-major format.)

Pure supports both *numeric* and *symbolic* matrices. The former use an internal representation which is compatible with the GNU Scientific Library (GSL); they must be homogeneous and contain either integer, floating point or complex values only. The latter can contain any mixture of Pure expressions (including other numeric or symbolic matrices). 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 (this also includes the case of bigints, which are considered as symbolic values as far as matrix construction is concerned).

Note that `{...}` isn't an ordinary constructor operation and so cannot be used for pattern-matching purposes. However, the usual size, indexing and slicing operations are all available, and the latter work both with 1- and 2-dimensional subscripts. 1-dimensional indices treat the matrix as a flat vector stored in row-major format, while

2-dimensional indices let you access the individual rows and columns of a matrix. Arbitrary ranges of these can be used with the slicing operation. Again, this works pretty much like in MATLAB and Octave, except that indices are zero- rather than one-based. Examples:

```
> let x = {1,2,3;4,5,6};
> #x; x!3; x!!(1..4);
6
4
{2,3,4,5}
> dim x; x!(1,2); x!!(0..1,1..2);
2,3
6
{2,3;5,6}
```

Note that `dim x` determines the dimensions (number of rows and columns) of a matrix, while `#x` just returns its number of elements. You can also retrieve the rows and columns of a matrix, as well as its main, sub- and super-diagonals:

```
> rows x; cols x;
[{1,2,3},{4,5,6}]
[{1;4},{2;5},{3;6}]
> row x 1; col x 1;
{4,5,6}
{2;5}
> diag x; subdiag x 1; supdiag x 1;
{1,5}
{4}
{2}
```

It is worth noting here that the operations to extract rows, columns and contiguous slices of matrices are optimized so that they won't copy any matrix elements, but reuse the underlying storage of the original matrix. This makes retrieving the rectangular parts of a matrix a very cheap operation, which is important for many numeric algorithms. Because of this, there may be a difference between the number of columns of a matrix and its *stride*, i.e., the actual row size of the matrix in memory:

```
> let y = x!!(0..1,1..2); dim y; stride y;
2,2
3
```

Usually you don't have to worry about the stride of a matrix, but this value may be important when passing matrices to C routines. You can also `pack` a matrix which copies its contents to fresh memory and makes sure that elements are stored consecutively:

```
> let z = pack y; dim z; stride z;
2,2
```

```
2
```

Another useful operation is *transposition*, which turns the rows of a matrix into its columns and vice versa. Note that transposing a matrix twice gives the original matrix.

```
> transpose x; transpose (transpose x)==x;
{1,4;2,5;3,6}
1
```

Pure does *not* provide any built-in support for linear algebra algorithms and other advanced matrix operations, but the facilities for matrix and list processing make it easy to roll your own, if desired. In particular, the prelude provides matrix versions of the common list operations like `map`, `fold`, `zip` etc. which can be employed for that purpose. E.g., multiplying a matrix `x` with a scalar `a` amounts to mapping the function `\x->a*x` on `x`, which can be done as follows:

```
> a * x::matrix = map (a*) x;
> 2*{1,2,3;4,5,6};
{2,4,6;8,10,12}
```

Moreover, it's possible to pass numeric matrices directly to GSL routines for doing numeric computations. You can also get at the underlying storage pointer of a matrix in order to shovel around matrix and vector data between Pure and third-party libraries written in C, where they may represent, e.g., raw pixel or audio data. This turns Pure into a useful tool for signal processing and other multimedia applications. We'll explore some of these possibilities in Chapter 8.

### 4.8.7 Records

*Records* are just symbolic vectors whose members are "hash pairs" of the form `key=>value`. Keys may be symbols or strings. For instance, `{x=>5,y=>12}` denotes a record value with two fields `x` and `y` bound to the values `5` and `12`, respectively. The field values can be any kind of Pure data. In particular, they may themselves be records, so records can be nested, as in `{x=>5,y=>{a=>"foo",b=>12}}`. The prelude provides various operations on records which let you retrieve field values by indexing and perform non-destructive updates. For instance:

```
> 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 = {x=>5,y=>{a=>"foo",b=>12}}; // nested record
> r!x,r!y,r!y!a;
5,{a=>"foo",b=>12},"foo"
```

The standard library provides a number other useful operations, please see the Pure Library Manual for details. Also, since Pure's records are just symbolic vectors, the full range of generic matrix operations (including matrix comprehensions, see below) is applicable to these objects. This turns them into a very versatile data structure, much more powerful than records in conventional programming languages which are usually limited to constructing records and accessing or modifying their components.

### 4.8.8 Comprehensions

Pure provides the usual comprehension syntax as a convenient means to construct both list and matrix values from a *template* expression and one or more generator and filter clauses. The clauses are considered from left to right. *Generator clauses* take the form $p$ `=` $x$ and bind the variables in the pattern $p$ to values drawn from a list or matrix $x$. Each generator introduces a new nested scope, similar to a **`when`** expression, but note that here the pattern $p$ gets matched to all members of the list or matrix value $x$ in turn. *Filter clauses* are just normal expressions. They must return a truth value which determines which generated elements should actually be included in the result list or matrix. For instance:

```
> [x,y | x=1..3; y=1..4; x<y];
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Comprehensions are really just syntactic sugar for combinations of lambdas, conditional expressions and various list and matrix operations. The interpreter expands these expressions at compile time. Specifically, list comprehensions are essentially implemented according to the following equivalences:[6]

| | | |
|---|---|---|
| `[` $x$ `\|` $y = z$ `]` | $\equiv$ | `map (\` $y$ `->` $x$ `)` $z$ |
| `[` $x$ `\|` $y = z$`;` *clauses* `]` | $\equiv$ | `catmap (\` $y$ `->` `[` $x$ `\|` *clauses* `]) ` $z$ |
| `[` $x$ `\|` $p$`;` *clauses* `]` | $\equiv$ | **`if`** $p$ **`then`** `[` $x$ `\|` *clauses* `]` **`else`** `[]` |

Here, `catmap` combines `cat` (which concatenates a list of lists) and `map` (which maps a function over a list). These operations are all defined in the prelude. Example:

[6] The actual implementation is slightly more complicated, in order to properly deal with different cases of generators.

```
> foo n m = [x,y | x=1..n; y=1..m; x<y];
> show foo
foo n m = catmap (\x -> catmap (\y -> if x<y then [(x,y)] else []) (1..m))
(1..n);
```

Matrix comprehensions work in a similar fashion, but with a special twist. If a matrix comprehension draws values from several lists, it alternates between row and column generation. (More precisely, the last generator, which varies most quickly, always yields a row, the next-to-last one a column of these row vectors, and so on.) 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 here):

```
> eye n = {i==j | i = 1..n; j = 1..n};
> eye 3;
{1,0,0;0,1,0;0,0,1}
```

Furthermore, if a matrix comprehension draws values from another matrix, it preserves the block structure of the input matrix:

```
> a*X::matrix = {a*x|x=X};
> 2*eye 2;
{2,0;0,2}
> {a*x|a={1,2;3,4};x=eye 2};
{1,0,2,0;0,1,0,2;3,0,4,0;0,3,0,4}
```

In any case, the result of a matrix comprehension must be something rectangular (which is always guaranteed if there are no filter clauses), otherwise evaluating the comprehension will throw a `bad_matrix_value` exception.

Comprehensions allow you to formulate many kinds of simple iterative algorithms which would typically be done using **`for`** loops in traditional programming languages, without having to resort to an imperative coding style. More elaborate examples for this very useful construct are discussed in Chapter 8.

## 4.9 Evaluation Order and Special Forms

Pure normally evaluates expressions using *call-by-value*, i.e., all subexpressions of an expression are evaluated before the expression itself. This is described in more detail in Chapter 6. However, some operations are actually implemented as *special forms* which defer the evaluation of some or all of their arguments until they are needed (i.e., doing *call-by-name* evaluation). In particular, the conditional expression **`if`** *x* **`then`** *y* **`else`** *z* 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*. Similarly, the logical connectives `&&` and `||` evaluate their operands in short-circuit mode. Thus, e.g., *x* `&&` *y* immediately becomes false if *x* evaluates to false, without ever evaluating *y*. Otherwise, *y* is evaluated and returned as the result of the expression.

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 (cf. Section 4.6), which also allows you to execute a function solely for its side-effects and just ignore the return value (this is explained in Section 5.2):

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

Other special forms are the function `catch` which handles exceptions, and the postfix operator `&` which does lazy evaluation in Pure. These will be described in Sections 7.2 and 7.3, respectively.

Last but not least, the special form `quote` quotes an expression, i.e., `quote` $x$ returns just $x$ itself without evaluating it. For convenience, this can also be written as `'`$x$. The prelude 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 facility should be well familiar to Lisp programmers, but there are some notable differences. First, only simple expressions can be quoted in Pure, special constructs such as conditionals, lambdas and scope expressions (**`case`**, **`when`** and **`with`**) will always be evaluated as usual. Second, local variables can't be quoted either, they will always be substituted, even in a quoted expression. For instance:

```
> '(2*42+2^n) when n = 2*6 end;
2*42+2^12
> '(2*42+(2^n when n = 2*6 end));
2*42+4096.0
```

On the other hand, the quote does inhibit evaluation inside matrix values, including the "splicing" of embedded submatrices:

```
> '{1,2+3,2*3};
```

```
{1,2+3,2*3}
> '{1,{2,3},4};
{1,{2,3},4}
```

Finally, note that 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.

Chapter 5

# Definitions

A Pure program may contain any number of *equations* a.k.a. *rewriting rules* which are used to define *functions* and *macros*. For convenience, Pure also supports *constant* and *variable* definitions. These elements take the following forms (cf. Appendix A):

*lhs* `=` *rhs*`;` Rewriting rules always consist of a left-hand side pattern *lhs* (which must be a simple expression, cf. Section 4.3) and a right-hand side *rhs* (which can be any kind of Pure expression as described in the previous chapter). There are some variations of the form of rewriting rules (guards, multiple left- and right-hand sides) which will be discussed in Section 5.2 below.

**`def`** *lhs* `=` *rhs*`;` This is a special form of rewriting rule used to expand *macro definitions* at compile time.

**`let`** *lhs* `=` *rhs*`;` This kind of definition binds every variable in the left-hand side pattern to the corresponding subterm of the right-hand side (after evaluating the latter). 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*`;` This is 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, cf. Section 4.3.) Unlike variables, **`const`** symbols can only be defined once, and thus their values do not change during program execution.

Macro definitions will be discussed in Section 7.4. The other kinds of definitions are described in the following, after some general remarks about the global scope and the rule syntax. Finally, we also explain how Pure allows you to organize your definitions into separate modules.

## 5.1 The Global Scope

In difference to local functions and variables introduced with **`with`** and **`when`** (cf. Section 4.6), the constructs discussed above define symbols with *global scope*. To facilitate interactive usage, the global environment is handled a bit differently from local scopes, in that the scope of each global definition extends from the point where the function, macro, variable or constant is first defined, to the next point where the symbol is redefined in some way. This makes it possible, e.g., to redefine global variables at any time:

```
> foo x = c*x;
> foo 99;
c*99
> let c = 2; foo 99;
198
> let c = 3; foo 99;
297
```

Similarly, you can also refine your function definitions as you go along. The interpreter automatically recompiles your definitions as needed when you do this. For instance:

```
> bar x = x if x>=0;
> bar 1; bar (-1);
1
bar (-1)
> bar x = -x if x<0;
> bar 1; bar (-1);
1
1
```

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. 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. This means that, even in a script file, you have to define all symbols needed in an evaluation *before* entering the expression to be evaluated.

## 5.2 Rule Syntax

All global and local definitions in Pure share the same kind of basic rule syntax, which consists of a left-hand side expression *lhs* and a right-hand side expression *rhs*, separated by the reserved = symbol. That is: *lhs* = *rhs*.

The left-hand side of a rule is a special kind of "template" expression called a *pattern*. Please refer to Section 4.3 for a closer description of these. To briefly recall the most important features, patterns must be simple expressions with no repeated variables (other than the anonymous variable '_'), which may also contain the following special elements:

- A left-hand side variable (including the anonymous variable) may be followed by one of the special type tags `::int`, `::bigint`, `::double`, `::string`, `::matrix` and `::pointer`, to indicate that it can only match a constant value of the corresponding built-in type.
- An "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).

In some cases (specifically, global and local function definitions as well as **`case`**), rules can also be augmented with a *guard*, and repeated left-hand or right-hand sides can be "factored out", as described below. Also note that in the case of function and macro definitions, as well as in **`case`** expressions, 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. (The **`let`**, **`const`** and **`when`** constructs are treated differently, as each rule is a separate definition.)

### 5.2.1 Guards

A guard is simply a condition of the from '**`if`** *expr*' tacked on to the end of a rule. The condition must evaluate to a machine integer; otherwise a `failed_cond` exception is raised. The rule is only applicable if the condition yields a nonzero value (which, as usual, means "true"). For notational convenience, the keyword **`otherwise`** can be used to indicate an empty guard which is always "true". This is just syntactic sugar for the human reader and is treated as a comment by the compiler; you can always omit it. For instance, consider the following definition of the factorial:

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

Here the first rule handles the case that the argument of `fact` is positive; in all other cases this rule will be ignored and the second rule will be invoked instead.

### 5.2.2 Repeated Left-Hand and Right-Hand Sides

In function definitions and **`case`** expressions, 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, the above definition of the factorial can also be written as follows:

```
fact n = n*fact (n-1) if n>0;
       = 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 if you need different specializations of the same rule which use different type tags on the left-hand side variables. For instance:

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

The above definition expands to two equations which share the right-hand side of the second equation, as if you had written:

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

This is essentially equivalent to just the last equation, but the "special case rule" for the `n::int` argument allows the compiler to generate more efficient code for that case by inlining machine code for the arithmetic and relational operations, because it is already known that the argument is a machine integer.

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

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

However, this is most useful when using an "as" pattern to bind 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 `sums`, if it is either the empty list or a list starting with an integer:

```
sums xs@[] | sums xs@(_::int:_) = scanl (+) 0 xs;
```

The same construct 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;
```

### 5.2.3 Local Definitions in Rules

Sometimes it is useful if local definitions (**when** and **with**) can be shared by the right-hand side and the guard of a rule in a function definition or a **case** expression. 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 $x^2 + px + q = 0$ 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.

### 5.2.4 Optional Left-Hand Sides

The **when**, **let** and **const** constructs only use the basic rule syntax which just consists of a left-hand and a right-hand side separated by '='. Guards or multiple left-hand or right-hand sides are not permitted in these. 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 debugging purposes. For instance, here is a variation of the quadratic equation solver from above 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
```

## 5.3 Function Definitions

In Pure, functions are defined by a collection of equations, using the rule syntax described above. The left-hand side pattern can be either a function by itself, or a function applied to some parameters. For instance, the following equation defines a function `square` which squares its argument by multiplying it with itself:

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

For now you'll have to take these at face value; in Chapter 6 we'll describe how exactly these equations are used to rewrite expressions. Of course, definitions may also be recursive, like the following definition of the Ackerman function:

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

Definitions may consist of any number of rules, which are considered in the order in which they are written. Here is another definition of the Ackerman function, which splits the conditionals in the definition above into three separate equations, two guarded rules for the `x==0` and `y==0` cases, and a "default" rule which applies to all other cases:

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

Structured arguments may be deconstructed using pattern matching. E.g., a function summing up all values in a list can be defined recursively as follows:

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

This also works with user-defined constructors. 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 the `nil` symbol needs to be declared as a nonfix symbol here, so that the compiler doesn't mistake it for a variable. The following example illustrates how the above definition may be used to obtain a binary tree data structure from a list:

```
> foldl insert nil [7,12,9,5];
bin 7 (bin 5 nil nil) (bin 12 (bin 9 nil nil) nil)
```

Functions can be *higher order*, i.e., they may take functions as arguments or return them as results. For instance, the `foldl` function, which generalizes the above `sum` function to accumulate all members of a list using any binary function `f` and any start value `a`, is defined in the prelude as follows:

```
foldl f a []     = a;
foldl f a (x:xs) = foldl f (f a x) xs;
```

Using this function, `sum` can be rewritten as follows:

```
sum = foldl (+) 0;
```

This is also called "pointless" style (pun intended), since it doesn't explicitly mention the function argument. Also note that this definition is not quite equivalent to the previous one; the previous definition accumulated results from right to left rather than from left to right, as `foldl` does. The prelude also provides another accumulation function `foldr` if you prefer the former.

Note that since operators are just function symbols in disguise, they can be used on the left-hand side of equations just as well. For instance, here is how you can define a lexicographic order on lists (of course, you'll have to add more equations for the other relational operators, but these are completely analogous):

```
[]   <= []   = 1;
[]   <= y:ys = 1;
x:xs <= []   = 0;
x:xs <= y:ys = x<=y && xs<=ys;
```

In Pure there is no need to specify all equations for a given function in one go; they may actually be scattered out through your program, and even over different source files (cf. Section 5.5). Thus the definition of a function can be refined at any time, and it can be as polymorphic (apply to as many types of arguments) as you like. The only requirement that is checked by the compiler is that all equations for a given function must agree on the number of function arguments. It will also warn you if there are any "useless" equations which are "shadowed" by previous definitions (which is usually a programming error). For instance:

```
> foo (x:xs) = x,xs;
> foo [x] = x;
> foo [1];
```

```
warning: rule never reduced: foo [x] = x;
1,[]
```

Here the programmer didn't realize that the first equation for `foo` already subsumes the singleton list case of the second equation. This kind of slip occurs more often than you might think, especially with complicated definitions involving a lot of equations. The compiler helps you identifying those errors. (This only works in simple cases where no guards are involved. The problem to identify "shadowed" guarded equations is in general undecidable.)

Also note that the compiler warning was only produced when we evaluated an expression after entering the equations. This is because the Pure interpreter compiles your definitions incrementally and "just in time", when they are needed. If you want to force immediate recompilation after some equations, it's sufficient to enter any expression like, say, `'1'`. This doesn't actually generate any executable code (this happens when a function is first used), but the compiler at least processes the definitions, generates the corresponding LLVM pseudocode and updates the internal symbol tables accordingly. At the interactive command line you can also use `Tab` completion for that purpose; just hitting the `Tab` key on a blank command line will run the compiler as well.

## 5.4 Constant and Variable Definitions

Term rewriting doesn't actually provide any means to define global "variables" which let you store computed values, but these are convenient for a number of different purposes. Therefore Pure has two closely related constructs which allow you to assign values to symbols occuring "free" in expressions: **`let`** which binds global variables, and **`const`** which binds constants. In difference to parameterless functions or macros, the bound value is only computed *once*, at the time the definition is processed. This is useful, in particular, to memoize values whose computation may be costly or involve functions with side-effects (such as opening a file).

The **`let`** construct is also commonly used interactively to bind variable symbols to intermediate results so that they can be reused later, e.g.:

```
> let x = 23/14; let y = 5*x; x; y;
1.64285714285714
8.21428571428571
```

Similar to local variable bindings in **`when`** clauses, the left-hand side of the definition may actually be a pattern to be matched against the computed value of the right-hand side. For instance, having defined the two global variables `x` and `y`, we might want to swap their values now, which can be simply done as follows:

```
> let y,x = x,y; x; y;
8.21428571428571
1.64285714285714
```

Constant definitions work in the same fashion, but the corresponding symbols can only be defined *once*. This is typically used for mathematical or physical constants. E.g.:

```
> const c = 299792.458;         // the speed of light, in km/s
> const ly = 365.25*24*60*60*c; // the length of a lightyear, in km
> ly;
9460730472580.8
```

Note that the right-hand side of a constant definition may be an arbitrary expression whose value is computed when the definition is processed, just as with **let**. But, since the value of a constant never changes once it is defined, it is possible to substitute the value *directly* into subsequent definitions, so that it doesn't have to be looked up at runtime:

```
> lys x = x*ly; // the length of x lightyears
> show lys
lys x = x*9460730472580.8;
```

We mention in passing here that the standard library already defines a bunch of **const** symbols for different purposes. E.g., the prelude contains the following definitions for the truth values:

```
> show true false
const false = 0;
const true = 1;
```

Finally, it is important to note the difference between **const** and **nonfix** symbols. The former are just *names*, placeholders for the actual values that are associated with them, while the latter are *real* symbolic constants (i.e., normal forms) which stand for themselves. In particular, **const** symbols will *not* work inside patterns (they will be treated as ordinary variables), unless you also declare them as **nonfix**. If you do this, the compiler substitutes their values also on the *left-hand side* of rules, which allows them to be matched, e.g., in **case** expressions as usual:

```
> nonfix true false;
> foo x = case x of true = "true"; false = "false"; _ = "???" end;
> show foo
foo x = case x of 1 = "true"; 0 = "false"; _ = "???" end;
> foo true, foo false, foo 99;
"true","false","???"
```

Note that declaring a **const** symbol **nonfix** makes it "precious"; see the caveats in Section 4.3. Therefore constant symbols are never declared **nonfix** in the library; if you want this, you'll have to do it yourself.

## 5.5 Programs and Modules

At the toplevel, a Pure program is basically just a collection of definitions, symbol declarations and expressions to be evaluated. However, as programs get bigger, you will want to partition them into separate modules. At this time, Pure doesn't support separate compilation, but it is possible to break down a program into a collection of source modules. Moreover, Pure provides a simple but effective namespace facility which lets you avoid name clashes between symbols of different modules and keep the global namespace tidy and clean.

Each module is just an ordinary Pure script. A special kind of **`using`** declaration is used to glue everything together. In particular, this declaration allows you to import definitions from standard library modules other than the prelude. For instance, the `sqrt` function resides in the `math` module, so if you want to use that function, you need to add the following declaration to your program (or type it at the interactive command line):

```
using math;
```

This actually *includes* the source of the `math.pure` script at this point in your program. Each module is included only *once*, at the point where the first **`using`** declaration for the module is encountered. The entire Pure program is then just the concatenation of the prelude and the scripts specified on the command line, with the modules listed in **`using`** clauses included in the appropriate locations.

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:

```
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.

Modules are first searched for in the directories of the scripts that use them; failing that, the interpreter also looks in the Pure library directory and some other "include directories" which may be configured with environment variables and/or command line options of the interpreter; please see the Pure manual for details.

### 5.5.1 The Standard Prelude

The *prelude* is actually an entire collection of modules which together form the core of Pure's standard library. You can find all the library modules in the Pure library directory

(usually `/usr/lib/pure` or `/usr/local/lib/pure` on Unix systems). The main script of the prelude is `prelude.pure` which imports the other prelude modules. Together, these modules implement the basic integer, list, tuple, string and matrix operations. The `prelude.pure` module is always loaded by the interpreter, so that the basic operations are available at the interactive command line and in all your programs.

Without the prelude, the interpreter provides little more than the basic term rewriting machinery and a few tie-ins for primitives which directly translate into native code. It is actually possible to start the interpreter without loading the prelude, by specifying the `-n` option when invoking the interpreter. This is useful if you absolutely need to start from the "bare metal" and define everything yourself, but is not recommended for normal usage.

### 5.5.2 Namespaces

All modules in your program share one global namespace, the *default* namespace, which is where new symbols are created by default, and which also holds most of the standard library operations. As your programs become larger, this bears the danger that the same symbol is used for different purposes in different modules, which may produce *name clashes* when the modules are linked together. To avoid this mischief, Pure allows you to put symbols into different user-defined namespaces. Like in C++, namespaces are completely decoupled from modules. It is possible to equip each module with its own namespace, but you can also have several namespaces in one module, or namespaces spanning several modules. The latter is useful, in particular, to handle collections of modules forming a *library*.

New namespaces are created with the **`namespace`** declaration, which also switches to the given namespace (makes it the *current* namespace), so that subsequent symbol declarations create symbols in that namespace rather than the default one. The current namespace applies to all kinds of symbol declarations, including fixity and **`extern`** declarations (the latter are described in Section 7.5). For instance, in order to create two symbols with the same print name `foo` in two different namespaces `foo` and `bar`, you can write:

```
namespace foo;
public foo;
foo x = x+1;
namespace bar;
public foo;
foo x = x-1;
namespace;
```

The **`public`** keyword makes sure that the declared symbols are visible out of their "home" namespace. (You can also declare symbols as **`private`**, see Section 5.5.3 below.) Also note that just the **`namespace`** keyword by itself in the last line switches back to the

default namespace. We can now refer to the symbols we just defined using qualified symbols of the form *namespace*`::`*symbol*:[1]

```
> foo::foo 99;
100
> bar::foo 99;
98
```

The namespace prefix can also be empty, to explicitly denote a symbol in the default namespace. (This is actually a special instance of an "absolute" namespace qualifier, to be explained below.)

```
> ::foo 99;
foo 99
```

This avoids any potential name clashes, since the qualified identifier notation always makes it clear which namespace the given identifier belongs to. However, as it is rather inconvenient if you always have to write identifiers in their fully qualified form, 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, as follows:

```
> using namespace foo;
> foo 99;
100
> using namespace bar;
> foo 99;
98
```

The **`using namespace`** declaration also lets you search multiple namespaces simultaneously:

```
using namespace foo, bar;
```

However, this requires that a symbol exists in at most one of the listed namespaces, otherwise you get an error message:

```
> using namespace foo, bar;
> foo 99;
<stdin>, line 15: symbol 'foo' is ambiguous here
```

In such a case you have to use the appropriate namespace qualifier to resolve the name clash:

```
> foo::foo 99;
100
```

[1] Note that a construct like `foo::int` may denote either a qualified identifier or a tagged variable (see Section 4.3.5) in Pure. The compiler assumes the former if `foo` is a valid namespace identifier. You can place spaces around the `::` symbol if this is not what you want. Since spaces are not allowed in qualified identifiers, this makes it clear that you mean a tagged variable instead.

A **using namespace** declaration without any namespace arguments gets you back to the default empty list of search namespaces:

```
using namespace;
```

In general, the scope of a **namespace** or **using namespace** declaration extends from the point of the declaration up to the next declaration of the same kind. Moreover, the scope is always confined to a single source file, i.e., namespace declarations never extend beyond the current script, and thus each source module starts in the default namespace with an empty list of search namespaces.

The precise rules for looking up symbols are as follows. 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 global namespace, in that order. (This automatic lookup can also be bypassed by using an absolute namespace qualifier of the form `::`*namespace*`::`*symbol*, see Section 5.5.4 below.)

If no existing symbol is found, a new symbol is created, implicitly declaring the identifier as a public symbol with default attributes. New *unqualified* symbols are always created in the default namespace, unless you explicitly declare them (in which case they become members of the current namespace, as explained above). New *qualified* symbols are created in the given namespace, which *must* be the current namespace. This makes it possible to avoid explicit symbol declarations in the common case of ordinary, public identifiers. E.g., we could have written the above example simply as follows:

```
namespace foo;
foo::foo x = x+1;
namespace bar;
bar::foo x = x-1;
namespace;
```

As a little safety measure against silly typos, the compiler insists that new qualified symbols must be introduced in their "home" namespace, otherwise it complains about an undeclared symbol:

```
> namespace;
> foo::bar x = 1/x;
<stdin>, line 7: undeclared symbol 'foo::bar'
```

To avoid such errors, you have to make sure that the right namespace is current when introducing the symbol.

Explicit declarations are always needed if you want to introduce special operator and constant symbols. Declaring these in a specific namespace works just like declarations in the default namespace, except that you add the appropriate namespace declaration 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;
> public infixl 6 +;
```

```
> x+y = x*y;
> 5+7;
35
```

(The keyword **public** in front of **infixl** can also be omitted, since symbols are always public unless you explicitly declare them as private symbols, see Section 5.5.3.)

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
```

### 5.5.3 Private Symbols

Pure also allows you to have *private* symbols, as a means to hide away internal operations which shouldn't be accessed directly by client programs. The scope of a private symbol is confined to its namespace, i.e., the symbol is visible only if its "home" namespace is the current namespace. Symbols are declared private by using the **private** keyword (instead of **public**) 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` has become invisible, because we switched back to the default namespace. This holds even if you have `secret` in the search namespace list:

```
> using namespace secret;
> // this actually creates a new symbol 'baz' 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 `secret::baz` back into scope is to make the `secret` namespace current again:

```
> namespace secret;
> baz 99;
198
> secret::baz 99;
198
```

Note that you should only do this if you are actually the "owner" or "creator" of the namespace accessed with the **`namespace`** declaration. To enforce information hiding, client modules should always use the **`using namespace`** declaration to access a namespace. This hides away the private symbols so that you do not inadvertently use those symbols in an inappropriate manner. This restriction does *not* apply to the public symbols of a namespace, which are always visible so that you can extend the corresponding operations by adding to their definitions in the usual way.

### 5.5.4 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. For instance:

```
namespace my;
namespace my::old;
my::old::foo x = x+1;
namespace my::new;
my::new::foo x = x-1;
```

Note that the namespace `my`, which serves as the parent namespace, must be created before creating 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
```

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;
```

```
public foo;
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
```

(Note that the notation `::foo` mentioned earlier, which denotes a symbol `foo` in the default namespace, is just a special instance of this notation for the case of an empty namespace qualifier.)

Chapter 6

# Rewriting

We still have to discuss how all those pretty function definitions are actually used to evaluate expressions. In Pure this is done using an abstract model of computation, *term rewriting*, which is also the workhorse behind computer algebra systems and theorem provers. Term rewriting is considerably more general in some ways than the lambda calculus (which is the backbone of most other functional languages). Its use as a programming language was first explored by Michael O'Donnell [15]. Pure implements term rewriting in a very efficient way, essentially compiling it down to ordinary function definitions in native code which can be executed directly by the cpu of your computer.

## 6.1 Term Rewriting in a Nutshell

Up to now, we have used the notions of expressions and rewriting rules in a fairly informal manner, so for a change let's take a brief look at the formal background of term rewriting theory. For the sake of simplicity, we discuss the theoretical framework using simple, unconditional rewriting rules; the treatment of guarded equations is left to the following section.

Here are some convenient definitions. A *signature* is a set $\Sigma = \biguplus_{n\geq 0} \Sigma_n$ of function and variable symbols. If $f \in \Sigma_n$ then we also say that $f$ has *arity* $n$, and we assume that $X_\Sigma \subseteq \Sigma_0$, where $X_\Sigma$ is the set of all variable symbols in $\Sigma$. The (free) *term algebra* over the signature $\Sigma$ is the set of terms defined recursively as $T_\Sigma = \{f\, t_1 \cdots t_n \mid f \in \Sigma_n, t_i \in T_\Sigma\}$.[1]

A *term rewriting rule* is a pair of terms $p, q \in T_\Sigma$, commonly denoted $p \to q$. In order to describe the meaning of these, we also need the notion of a *substitution* $\sigma$ which is simply a mapping from variables to terms, $\sigma : X_\Sigma \mapsto T_\Sigma$. For convenience, we also write these as $[x_1 \to \sigma(x_1), x_2 \to \sigma(x_2), \ldots]$, and we assume that $\sigma(x) = x$ unless ex-

[1] Note that term rewriting theory usually employs uncurried function applications, but the curried notation can actually be seen as a special case of these, where all function symbols are nullary, except for one binary symbol which is used to denote function application.

plicitly mentioned otherwise. Given a term $p$ and a substitution $\sigma = [x_1 \to \sigma(x_1), x_2 \to \sigma(x_2), \ldots]$, by $\sigma(p) = p[x_1 \to \sigma(x_1), x_2 \to \sigma(x_2), \ldots]$ we denote the term obtained by replacing each variable $x$ in $p$ with the corresponding $\sigma(x)$. For instance, if $p = f\,x\,y$ then $p[x \to g\,x, y \to c] = f\,(g\,x)\,c$. We also say that a term $u$ *matches* a term $p$, or is an *instance* of $p$, if there is a substitution $\sigma$ (the so-called *matching substitution*) such that $\sigma(p) = u$.

A *context* in a term $t$ is a term $s$ containing a single instance of the distinguished variable $\omega$ such that $t = s[\omega \to u]$. That is, $t$ is just $s$ with the subterm $u$ at the position indicated by $\omega$.

Now the stage is set to describe an application of a term rewriting rule $p \to q$ to a subject term $t$, given a context $s$ in $t$. Suppose that $t = s[\omega \to u]$, where $u = \sigma(p)$. Then we can rewrite $t$ to $t' = s[\omega \to v]$ where $v = \sigma(q)$. Such a single rewriting step is also called a *reduction*, and $u$ and $v$ are called the *redex* and the *reduct* involved in the reduction, respectively. For instance, by applying the rule $f\,x\,y \to h\,x$ to the subject term $t = f\,(g\,x)\,c$, where the context is just $s = \omega$ and the matching substitution is $[x \to g\,x, y \to c]$, we obtain $t' = h\,(g\,x)$.

Term rewriting rules are rarely applied in isolation, they usually come in droves, called *term rewriting systems*. Formally, a term rewriting system is a finite set $R$ of term rewriting rules. We write $t \to_R t'$ if $t$ reduces to $t'$ by applying any of the rules $p \to q \in R$, and $t \to_R^* t'$ if $t$ reduces to $t'$ using $R$ in any number of single reduction steps (including zero). That is, $\to_R^*$ is the reflexive and transitive closure of the single step reduction relation $\to_R$. Similarly, $\leftrightarrow_R^*$ is the reflexive, transitive *and* symmetric closure of $\to_R$.

Finally, a term $t$ is said to be *irreducible* or in *normal form* (with respect to $R$) if no rule in $R$ applies to it, i.e., there is *no* term $t'$ such that $t \to_R t'$. If $t \to_R^* t'$ such that $t'$ is in normal form, then we also call $t'$ a *normal form of* $t$.

So what is the use of this in mathematical logic? Given a term rewriting system $R$, we can also look at the corresponding set of equations $E = \{p = q \mid p \to q \in R\}$. As it turns out, any "model" which satisfies the equations in $E$ (taken variables to be universally quantified) will also satisfy all equations given by $\leftrightarrow_R^*$. Under certain circumstances, the rewriting system can then be used as a procedure for deciding whether two given terms are "equal" by just comparing their normal forms.

Note that normal forms need not always exist (rewriting a given term $t$ might not terminate, because there's always yet another rule which can be applied) and even if they do, they might not be unique (different reduction sequences might yield different normal forms of the same subject term). In fact these questions are not even decidable, because term rewriting is Turing-complete. That's why practical applications of term rewriting usually impose a suitable "reduction strategy", which is also the approach taken in Pure.

## 6.2 The Evaluation Process

So for our purposes term rewriting provides a way to evaluate an expression by reducing it to a normal form using the equations making up a Pure program, where we consider equations as rewriting rules, by orienting them from left to right. The normal form is then taken to be the "value" of the original expression. Note that we don't talk about "functions" here at all. All the Pure interpreter ever worries about are the expressions to be evaluated and the equations which can be applied to them.

Because normal forms are not necessarily unique, we need to impose a certain *reduction strategy* which determines which redices are reduced in each step, and the rewriting rule to be applied in order to produce the reduct. In Pure, expressions are normally evaluated using the *leftmost-innermost* reduction strategy. That is, expressions are evaluated from left to right, innermost expressions first. This implies that in a function application, first the function object is evaluated, then its argument, and finally the function is applied to the argument. This is also known as *call-by-value*. (Pure also provides means to do *call-by-need* evaluation, but we won't go into that here; see Section 7.3.)

For instance, let's consider the following simple definition: `square x = x*x;` This is to be read as the term rewriting rule `square x` → `x*x`. In order to evaluate `square (5+8)`, the interpreter then does the following reductions (this also invokes some rules for handling addition and multiplication which are provided in the prelude). The redex in each step is underlined:

$$\texttt{square (}\underline{\texttt{5+8}}\texttt{)} \rightarrow \underline{\texttt{square 13}} \rightarrow \underline{\texttt{13*13}} \rightarrow \texttt{169}.$$

Note that, because of call-by-value semantics, rules need to be written in a way so that they match *normal form* arguments. For instance, if a program contains two rules `foo (bar x) = x` and `bar x = x+1`, then the first rule will *never* be applicable, since any `bar x` subterm will already have been reduced by the second rule before the first rule can be applied.

Conceptually, for each reduction step the interpreter has to perform the following operations:

1. Match the subject expression against the left-hand sides of equations. If more than one equation matches, they are tried in the order in which they are listed in the program. If no equation matches, the expression is already in normal form and we're done.

2. Bind the variables in the matching equation to their corresponding values. (This amounts to constructing the matching substitution.)

3. For conditional equations, evaluate the guard using the variable binding determined in step 2. If the guard fails, try the next matching equation. Otherwise proceed with step 4.

4. Evaluate the right-hand side of the equation using the variable binding determined in step 2.

This might seem inefficient, but luckily the Pure interpreter compiles your program to fast native code before executing it. The pattern-matching code uses a kind of optimal decision tree which only needs a single, non-backtracking left-to-right scan of the subject term to determine *all* matching equations in one go. In most cases the matching overhead is barely noticable, unless you discriminate over large sets of heavily "overlapping" patterns, see [7] for details. Using these techniques and native compilation, the Pure interpreter is able to achieve very good performance, offering execution speeds in the same ballpark as good Lisp interpreters.

Note the choice of matching equations in step 1 of the evaluation procedure. If more than one equation matches, the reduction strategy must specify unambiguously the order in which they are to be tried. In Pure the equations are always considered in the *lexical* order, i.e., in the order in which they are written by the programmer. (There are other ways to do this, e.g., [17] proposes a "specifity" order, but the lexical order seems to be the most straightforward and intuitive to use.) Thus, if a function is defined by multiple, possibly ambiguous equations then the most specific equations should come first, as in the following definition of the factorial:

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

The first equation handles the `n>0` case, while the second equation is to be applied as a "default" rule when the guard of the first rule yields "false". So the evaluation of `fact 1` proceeds as follows:

$$\underline{\texttt{fact 1}} \overset{(1)}{\to} \texttt{1*fact (}\underline{\texttt{1-1}}\texttt{)} \to \texttt{1*}\underline{\texttt{fact 0}} \overset{(2)}{\to} \underline{\texttt{1*1}} \to \texttt{1}.$$

First `fact 1` is reduced to `1*fact (1-1)` by the first rule which is applicable here because `1>0` → `1`. But in the third reduction the guard `n>0` "fails" (`0>0` → `0`) and hence the second equation is applied to `fact 0`, which reduces it to `1`.

Together with leftmost-innermost evaluation, the lexical rule order removes all ambiguity in the evaluation process so that it becomes completely deterministic. The lexical order is convenient, but it also has some drawbacks. Specifically, you have to be careful if functions are defined by equations scattered out over different scripts (which is often the case for heavily polymorphic operations such as `+` and `==`). The compiler can warn you about cases where one equation is "shadowed" by another, but this only works for unguarded equations (cf. Section 5.3). Hence you should make sure that the scripts are always linked in the right order by judicious use of the **`using`** declaration (cf. Section 5.5).

## 6.3 Rewriting Rule Examples

The proof of the pudding is in the eating, so let's look at some examples illustrating how this all works out in practice. As we have seen, Pure programs are simply collections of equations which are used to rewrite expressions in a symbolic fashion. In the simplest case, such an equation looks just like an ordinary function definition:

```
> square x = x*x;
> square 99;
9801
```

Not very exciting so far. But Pure also allows you to apply this definition to *symbolic* inputs:

```
> square (a+b);
(a+b)*(a+b)
```

Moreover, the left-hand side of an equation can in fact be an arbitrary (simple) expression. For instance, here are some symbolic rewriting rules for associativity and distributivity of the + and * operators:

```
> (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
```

Note that rules like the above aren't possible in most functional languages, because they violate the "constructor discipline", which demands that only "free" constructors (i.e., function symbols without defining equations) should be used in the argument patterns. Of course, no one forbids you to do these simple kinds of "pattern-matching" definitions in Pure, so the customary programming techniques from languages like Haskell or ML carry over quite easily. In fact, we've already seen many examples of these; for instance, recall the definition of the `uniq` operation:

```
> uniq (x:x:xs) = uniq (x:xs);
> uniq (x:xs)   = x:uniq xs;
> uniq []       = [];
> uniq [1,2,2,3,3,3,1];
[1,2,3,1]
```

Definitions like the one above are just a special kind of the general rewriting rules that Pure offers. This makes the language conceptually much simpler, and at the same time gives you considerably more freedom in that you can also have algebraic simplification rules of the kind we've seen above. This might appear to be an arcane feature, but it's actually really useful in some situations. In particular, "constructor equations" *always* violate the constructor discipline, so they can't be used in Haskell or ML. For instance, suppose that we want lists to automatically stay sorted and eliminate duplicates. In Pure we can do this by simply adding the following equations:

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

You wouldn't really want to add such a definition to most programs since it would affect *all* list values. But similar definitions actually prove useful when defining custom data structures such as sets. Another real-world example is Pure's exact division operator `%` which also acts as a constructor for rational numbers in the `math` module. In this case the constructor equation

```
x::bigint%y::bigint = (x div d) % (y div d) if d>1 when d = gcd x y end;
```

takes care that rational numbers are always represented in lowest terms:

```
> using math;
> 48L%30L;
8L%5L
```

## 6.4 Dynamic Typing

Let us finally take a look at the issue of representing data structures in Pure. Like Lisp, Pure is a dynamically typed language, and there are no data type declarations, nor do we need them. In Pure, "data" means just normal form expressions, which all belong to the same universe of terms. In order to deal with different kinds of data, you therefore have to distinguish them with an appropriate choice of constructor symbols.

Custom data structures can be defined in Pure just as easily as the predefined list and tuple aggregates. You only have to pick some suitable constructor symbols. Normally, these are introduced on the fly; only constant symbols must be declared, if they are to be used on the left-hand side of equations. For instance, we've already mentioned binary trees which are useful for sorting and searching. These can be represented using the constructor symbols `nil` (a nonfix symbol denoting the empty tree) and `bin` (a ternary constructor denoting an interior node of the tree, which takes a data element and the left and right subtrees as arguments). The following definition then implements a binary tree insertion routine:

```
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;
```

Creating a tree from a list of elements can now be done as follows:

```
bintree = foldl insert nil;
```

Finally, to convert a tree back to an (ordered) list we just do an inorder traversal of the tree:

```
members nil         = [];
members (bin x L R) = members L + (x:members R);
```

We can use this, e.g., to sort a list as follows:

```
> members (bintree [7,3,9,18,3]);
[3,3,7,9,18]
```

So we just implemented a binary tree data structure in six simple equations and seven lines of code, not too bad. Of course, term rewriting is pretty much tailored to do these kinds of tree manipulations. Where it falls short is in dealing with variable-sized structures requiring efficient random element access. The best you get with plain term rewriting are balanced tree structures which require logarithmic running time for accessing individual elements. But Pure works around this limitation with its matrix data structure which provides for constant time access to its members.

By generalizing the above binary tree example, you can deal with pretty much any kind of "algebraic" or "variant record" data type, where the constructors are employed as variant selectors and pattern matching can be used to simultaneously discriminate over the variants and extract the component values. In addition, *abstract data types* can be realized by hiding the constructors (putting them into a special namespace and making them **`private`**), so that client modules must go through the provided public operations to access the data structure. E.g., in the example above all you have to do is add the following declaration at the beginning of the script:

```
namespace bintree;
private nil bin;
```

Dynamic typing has become a bit unusual in modern functional languages, which often employ a static type system with parametric polymorphism such as the Hindley/Milner system. The advantages of static typing are well-known. Static typing is safer because some kinds of programming errors can be caught at compile time rather than runtime. It also enables the generation of more efficient code because the types of arguments don't have to be checked at runtime. Some programmers also feel that the typing framework set by the language makes them think more about the data modelling aspects of their program designs, instead of inventing ad hoc data structures on the spot.

Nevertheless, using static typing in Pure would be a big mistake, because it would kill off most of the ad hoc polymorphism and symbolic manipulation capabilities offered by general term rewriting. Moreover, static typing is often perceived as rigid and inconvenient; in contrast, as we have seen above, Pure allows you to create new data structures almost without any effort. Also, as Oortmerssen argues [17, p. 11], there's no way that an input term can ever be "invalid" for a term rewriting system; either there's a rule which applies to it, which means that the term is reducible, or there isn't in which case the term is in normal form. Thus, since there are no dynamic "argument mismatch" errors, there can't be any static typing errors either. This provides some theoretical justification why dynamic typing is such a nice fit for term rewriting. The

situation is actually a bit more complicated in Pure because, e.g., failing guards and pattern matches may indeed raise exceptions, but in a dynamic language it still makes sense to treat these conditions as runtime errors which can be repaired by an application program, rather than forbidding such programs in the first place. The price you pay for the added flexibility is that Pure is inherently less safe than statically typed languages. But if you want Haskell or ML, you know where to find them.

Chapter 7

# Advanced Topics

While term rewriting is a universal model of computation in itself, as a practical programming language Pure also provides various enhancements which go beyond the basic rewriting machinery described in the previous chapter. In the following we explain how to implement iterative algorithms efficiently using tail recursion, how to handle various kinds of runtime errors, and how to deal with infinite data structures using lazy evaluation. We also discuss how you can extend the built-in capabilities of the interpreter with macros and by interfacing to the C programming language. Finally, we show you how you can turn your Pure scripts into standalone native executables using Pure's batch compiler.

## 7.1 Tail Recursion

Alan Perlis once said that "a program without a loop and a structured variable isn't worth writing." In fact, computers excel at carrying out repetitive tasks which would be impossible or at least very inconvenient if we had to do them manually. Thus carrying out the same or similar calculations with a large amount of data is an essential capability for any programming language.

In contrast to imperative programming languages, functional programming essentially relies on a single construct for repeating calculations: *recursion*. By this we mean that a function invokes itself, either directly or indirectly. As we will see, other repetitive control constructs can always be expressed in terms of recursion.

A well-known example of a repetitive calculation already discovered by the ancient Greeks is Euclid's algorithm for calculating the *greatest common divisor* of two integers. That is, given two (nonnegative) integers $x$ and $y$, we are looking for the largest $z$ such that $x$ and $y$ are both integral multiples of $z$. Euclid's solution to this problem rests on the observation that if $z$ divides both $x$ and $y$, then the same holds for $y$ and $x \bmod y$ (given that $y > 0$), where $x \bmod y$ denotes the remainder of the division of $x$ by $y$. Moreover, if $y$ is zero, then $z = x$ is the solution. This readily translates to the following

recursive procedure. (Note that we declare `gcd` in a special namespace here, because the prelude already defines a `gcd` function. Also, for the sake of simplicity, we assume that the parameters `x` and `y` are always nonnegative integers.)

```
namespace my;
public gcd;
gcd x y = gcd y (x mod y) if y>0;
        = x otherwise;
```

The algorithm terminates because the value of the second parameter is reduced in each recursive application. This can be seen if we evaluate an expression like `gcd 25 35`. (Here and in the following, for the sake of clarity we omit reductions for the arithmetic primitives.)

```
gcd 25 35 → gcd 35 25 → gcd 25 10 → gcd 10 5 → gcd 5 0 → 5
```

Note that the recursive invocation of `gcd` is always the last call (the so-called *tail call*) on the right-hand side of the first equation in the definition above. Such definitions are called *tail-recursive*. This is a desirable property, because the Pure interpreter does *tail call elimination*.[1] We won't go into all the gory details, but here is a brief explanation: Computers execute function calls using a *stack* which records the return address, parameters and other local data of a function, called the *activation record* of a function call. With tail call elimination, the last call on the right-hand side of a function definition simply reuses the existing activation record. Thus with a tail-recursive definition, the recursive calls don't use up any additional stack space at all, and the entire algorithm executes in constant stack space.

In contrast, consider our previous definition of the factorial:

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

This definition is *not* tail-recursive, since the tail call of the first equation invokes the `(*)` function rather than `fact` itself. This also becomes apparent when looking at a typical reduction sequence:

```
fact 3 → 3*fact 2 → 3*(2*fact 1) → 3*(2*(1*fact 0))
       → 3*(2*(1*1)) → 3*(2*1) → 3*2 → 6
```

Note how the intermediate subterms first grow while the redices wander inwards as the computation progresses. This means that for large values of `n` this definition is in danger of running out of stack space. For instance, try the following. (Section 7.2 shows how you can set up the interpreter so that it deals with stack overflows in a more civilized manner.)

[1] This requires that the JIT backend of your LLVM version supports this kind of optimization, which, as of LLVM 2.3 and later, should be the case for most popular computer architectures. Also note that tail call optimization is *always* disabled if the interpreter is run with the `-g` option, in order to facilitate debugging. Please see the Pure manual for details.

```
> fact 1000000;
Segmentation fault
```

Fortunately, it is easy to rewrite most simple kinds of recursive definitions so that they become tail-recursive, by employing the so-called *accumulating parameter* technique. The idea is, quite simply, to carry around intermediate results in an extra parameter. For instance:

```
fact n = loop 1 n with
  loop p n = loop (p*n) (n-1) if n>0;
           = p otherwise;
end;
```

Now the calculation of `fact 3` proceeds as follows:

$$\texttt{fact 3} \to \texttt{loop 1 3} \to \texttt{loop 3 2} \to \texttt{loop 6 1} \to \texttt{loop 6 0} \to \texttt{6}$$

This works pretty much like a **`for`** or **`while`** loop in conventional programming languages, but doesn't require any special looping construct or mutable variables, which Pure doesn't have. There are other, more general forms of recursion which defy this easy kind of transformation, and more elaborate techniques such as *continuation passing* to deal with them. But the accumulating parameter technique is easy to implement and works well for any kind of algorithm where you'd use some kind of loop control structure in conventional programming languages.

To summarize, while Pure doesn't provide any built-in looping constructs, you can easily roll your own, using the techniques sketched out above. In fact, the prelude already provides a number of generic functions of this kind, such as `do`, `map` and `fold`, as well as list and matrix comprehensions. These are all implemented in a tail-recursive fashion. Moreover, conditional expressions (**`if`** $x$ **`then`** $y$ **`else`** $z$) are subject to tail call elimination in both branches, while the logical operators `&&` and `||` as well as the sequence operator `$$` are tail-recursive in their second operands.

## 7.2 Exceptions

"To err is human, to forgive divine." Computational processes can run into problematic situations just like humans. This may be due to bugs in the program, unanticipated input errors, or resource constraints such as limited stack space. The notion of *exceptions* has been invented to deal with such unfortunate situations and provide a way to escape from them. More generally, they can be used to bail out in the middle of a computation and implement non-local value returns.

Every Pure exception has a *value* which can be any (normal form) expression. For instance, the predefined nonfix symbols `stack_fault`, `failed_cond` and `failed_match` stand for the built-in error conditions of stack overflow, failing guards and pattern matches, respectively. Some prelude operations may generate exceptions of their own,

such as the `out_of_bounds` exception raised by the `'!'` operator when an index is outside of the permitted range. You can generate any kind of exception yourself by just passing the desired value to the `throw` primitive. For instance:

```
> throw hello_world;
<stdin>, line 1: unhandled exception 'hello_world' while evaluating
'throw hello_world'
```

Handling an exception is just as easy:

```
> catch error (throw hello_world);
error hello_world
```

The `catch` primitive is a special form which takes two arguments: an exception handler $h$ and the expression $x$ to be evaluated. The latter is a "call-by-name" argument which gets evaluated by `catch`. If everything goes as expected, the value of $x$ is returned as the value of the `catch` call. But if an exception happens while evaluating $x$, then the value $y$ of the exception is passed to the handler $h$ and the result of the application $h\,y$ is returned instead. Note that the exception handler can in fact be any Pure value, like a simple constructor as in the example above, but usually it is a function designed to deal with the error in some way (e.g., print an error message) and return a suitable replacement for the value of $x$. For instance:

```
> using system;
> catch error (throw hello_world)
> with error x = printf "Hey, I got a '%s' exception!\n" (str x) $$ 0 end;
Hey, I got a 'hello_world' exception!
0
```

As already mentioned, 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 patterns in a pattern-matching lambda abstraction, or a **`let`**, **`case`** or **`when`** construct. 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 1000000);
error stack_fault
```

Note that Pure doesn't do stack checks by default, so you'll have to set the `PURE_STACK` environment variable to get the latter kind of exception; otherwise programs will just crash the interpreter with a "segfault" or some similarly unpleasant error message. E.g., the following shell command should do the trick on Linux:

```
$ export PURE_STACK=4096
```

You might have to adjust that value, depending on how much stack space (in kilobytes) is actually available to application programs on your system. Also, it's probably a good idea to add this line to your shell startup file once you have found a suitable setting. This is explained in more detail under "Stack Size and Tail Recursion" in the Pure manual.

Exceptions also provide a way to handle asynchronous signals. When running interactively, most standard termination signals (`SIGINT`, `SIGTERM`, etc.) are set up during startup of the interpreter to produce corresponding Pure exceptions of the form `signal` *n* where *n* is the signal number. 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;
```

Last but not least, exceptions can also be used in non-error situations, to implement non-local value returns. For instance, here is a quick and dirty way to implement a function `find` which returns the first member of a list `xs` which satisfies a given predicate `p`, or `()` if no such element is found.

```
> find p xs = catch id (do check xs) with check x = throw x if p x end;
> find (<0) [1,17,-5,9];
-5
```

Here, the "exception handler" is just the identity function `id` defined in the prelude. The definition above assumes that `p` itself never throws an exception and always yields a proper truth value. This can be made more robust with an explicit comparison and by wrapping the call to `p` in its own `catch` clause; we leave this as an exercise to the reader.

## 7.3 Lazy Evaluation

As already mentioned, Pure provides the special form '`&`' to handle lazy evaluation. This is a postfix operator, written as *x*`&`, where *x* is an arbitrary Pure expression. The '`&`' operator binds stronger than any other operation except function application. 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. This is also known as *call-by-need*.

Futures can be employed to implement all kinds of lazy data structures in Pure. Special support is provided in the Pure prelude for lazy lists, which are also called *streams* in the functional programming literature (these are not to be confused with the

kind of streams provided by the C library, which are just disk files). A stream is a list with a thunked tail, which allows it to be infinite, or so huge that you'd never want to keep it in main memory in its entirety. Most list operations have been designed so that they work with these kinds of objects just as well as with ordinary "eager" lists.

A simple way to obtain an infinite stream is to create an arithmetic sequence with an infinite upper bound, for instance:

```
> let x = 1:3..inf; x;
1:#<thunk 0xb5d6ca88>
```

Note the special thunk object in the tail of the stream which hasn't been evaluated yet. We can force a finite part of the stream to be evaluated, e.g., by cutting a "slice" from it:

```
> x!!(0..10);
[1,3,5,7,9,11,13,15,17,19,21]
```

Because of memoization the generated elements are now readily available if we need to look at them again, which avoids unnecessary reevaluations:

```
> x;
1:3:5:7:9:11:13:15:17:19:21:#<thunk 0xb5d6ce18>
```

The prelude provides a number of other stream generation functions. For instance, infinite arithmetic sequences are actually created with the `iterate` function, which can also be used in more general ways, such as generating a sequence of powers of two:

```
> show iterate
iterate f x = x:iterate f (f x)&;
> let x = iterate (2*) 1; x!!(0..10);
[1,2,4,8,16,32,64,128,256,512,1024]
```

The `repeat` and `cycle` functions repeat the same element or subsequence ad nauseam:

```
> 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]
```

These can all be combined with the usual generic list functions such as `map`, `foldl`, `scanl`, `zip` etc. For instance:

```
> let x = zipwith (*) (cycle [1,-1]) (1..inf); x!!(0..10);
[1,-2,3,-4,5,-6,7,-8,9,-10,11]
```

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 0xb5d6d610>
```

```
> 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)]
```

Of course, streams can also be defined explicitly. For instance, the following is an implementation of Erathostenes' sieve which generates the infinite stream of all prime numbers:

```
primes = sieve (2..inf) with
  sieve (p:qs)  = p : sieve [q | q = qs; q mod p] &;
end;
```

Note the '&' on the tail of the sieve; this is what turns the sieve into a stream and keeps the `sieve` function from looping. For instance, let's count how many primes there are below 10000:

```
> #takewhile (<=10000) primes;
1229
```

And here is how you can print a stream of integers like the above (this will keep on printing forever; hit `Ctrl-C` when you get bored):

```
> using system;
> do (printf "%d\n") primes;
2
3
5
7
11
...
```

## 7.4 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 simple 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. In difference to these, macro rules start out with the keyword **`def`**, and only simple kinds of rules without any guards or multiple left-hand and right-hand sides are permitted.

Syntactically, a macro definition looks just like a variable or constant definition, using **`def`** in lieu of **`let`** or **`const`**, but they are processed in a different way. Macros are substituted into the right-hand sides of function, constant and variable definitions. 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 evaluated using call by value (i.e., macro calls in macro arguments are expanded before the macro gets applied to its parameters).

Here are some simple examples (please see the Pure manual for more). Our first example is the following macro definition from the prelude, which eliminates saturated instances of the right-associative function application operator '$':

```
def f $ x = f x;
```

This is a simple example of an optimization rule which helps the compiler generate better code. In this case, saturated calls of the $ operator (which is defined as an ordinary function in the prelude) are "inlined" at compile time. 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 optimize calls to an existing function, as in the previous example. As with ordinary functions, the number of parameters in each rule for a given macro must be the same. In this example the number of arguments is in fact the same as that of the corresponding function, but in general the macro may well have a different arity.

You can also 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. Note that 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).

For instance, the following rule defines a macro `timex` which employs the function `clock` from the `system` module 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;
> timex (count 1000000) with count n = if n>0 then count(n-1) else n end;
0.71,0
```

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!

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. For instance, here is a (simplified) Pure version of Lisp's quasiquote:

```
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;
```

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 the `f@_` in the second and third rule, which is an anonymous "as" pattern forcing the compiler to recognize `f` as a function variable, rather than a literal function symbol. This trick is explained in Section 4.3.6. Also 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 in Pure, 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)
```

Since Pure macros can be recursive, they 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 Section 7.2 above for information on how to deal with these by setting the `PURE_STACK` environment variable accordingly.

Also note that 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.

Pure macros also have their limitations. Specifically, the left-hand side of a macro rule must be a simple expression, just like in ordinary function definitions. This re-

stricts the kinds of expressions which can be rewritten by a macro. But Pure macros are certainly powerful enough for most common preprocessing purposes, while still being robust and easy to use.

## 7.5 C Interface

Accessing C functions from Pure programs is dead simple. You just need an **extern** declaration of the function, which is a simplified kind of C prototype. The function can then 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
```

Multiple prototypes can be given in one **extern** declaration, separating them with commas, and the parameter types can also be annotated with parameter names (these are effectively treated as comments by the compiler, so they serve informational purposes only):

```
extern double sin(double), double cos(double);
extern double tan(double x);
```

The interpreter makes sure that the parameters in a call match; if not, the call is treated as a normal form expression by default, which gives you the opportunity to extend the external function with your own Pure equations. For instance:

```
> sin 1;
sin 1
> sin x::int = sin (double x);
> sin 1;
0.841470984807897
```

Sometimes it is also necessary to access a C function under a different name. To these ends, Pure allows you to specify an *alias* under which the original C function is known to the Pure program. An alias is introduced by terminating the **extern** declaration with a clause of the form '= *alias*'. For instance, here is how the `system` module provides a portable interface to the `nanosleep` function. This function is actually implemented under the name `pure_nanosleep` in the Pure runtime, which abstracts away the platform-specific details of accessing a high-resolution function for timed waits:

```
extern double pure_nanosleep(double) = nanosleep;
nanosleep t::int | nanosleep t::bigint = nanosleep (double t);
```

In any case, the Pure name of the function is a symbol in the *current* namespace (at the point of the **extern** declaration), cf. Section 5.5.2.

The range of supported C types in the interface is somewhat limited right now (`void`, `bool`, `char`, `short`, `int`, `long`, `float`, `double`, as well as arbitrary pointer types, i.e.: `void*`, `char*`, etc.), but in practice these cover most kinds of calls that need to be done when interfacing to C libraries.[2] The precise rules for marshalling Pure objects to corresponding C types are explained in the Pure manual. Briefly, the C interface translates between Pure integers and bigints and C integer types in the obvious way, using sign extension or truncation as needed. Likewise, `float` and `double` are converted from/to Pure's double precision floating numbers, promoting single to double precision and vice versa as needed.

Concerning the pointer types, `char*` is for string arguments and return values which need translation between Pure's internal UTF-8 representation and the system encoding, while `void*` is for any generic kind of pointer (including strings, which are *not* translated when passed/returned as `void*`). The `expr*` pointer type can be used to pass through Pure expressions just as they are, without any kind of (un)boxing. The `dmatrix*`, `cmatrix*` and `imatrix*` types denote GSL-compatible `double`, `complex double` and `int` matrices; this allows you to pass numeric matrices and return them as results of GSL routines. For convenience, it is also possible to pass a numeric matrix for a `short*`, `int*`, `float*` or `double*` parameter. The required conversions are done automatically, on the fly, and the matrix data is copied to temporary storage in order to preserve value semantics. In addition, any kind of matrix (including symbolic matrices) can also be passed for a generic `void*` pointer. In this case no conversions are done and a pointer to the raw matrix data is passed, which allows the matrix to be modified in-place.

All other pointer types are effectively treated as `void*` right now, although in a future version the interpreter may keep track of the type names for the purpose of checking parameter types.

Pure leaves it to the LLVM runtime to actually resolve linkage to external C functions. The runtime first looks for symbols in the C library and Pure's runtime library. Thus all C library and Pure runtime functions are readily available in Pure programs. Functions in other (shared) libraries can be accessed with a special form of the **`using`** clause. For instance, if you have some C functions in a shared library named, say, `myutils.so`, the following declaration loads the library so that you can get access to these functions in Pure through subsequent **`extern`** declarations:

```
using "lib:myutils";
```

The interpreter locates dynamic libraries in a way similar to source scripts, using a separate search path which takes into account the directory of the script with the **`using`** clause, custom search paths as well as system-specific library paths. Moreover, the proper library suffix (like `.so` on Linux, `.dll` on Windows) is supplied automatically. Please refer to the Pure manual for details, and also have a look at the C interface examples included in the distribution.

[2] Note that `long` may be either a 32 or a 64 bit integer type, depending on the architecture. Pure also provides the synonyms `int8`, `int16`, `int32` for `char`, `short`, `int`, as well as `int64` to denote 64 bit integers.

If you need to interface to large C libraries, there's a separate `pure-gen` program available at the Pure website which makes this easier. This Pure script 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. More information about this can be found on the *Addons* wiki page of the Pure website.[3]

Finally, a word of caution: The interpreter always takes your **`extern`** declarations 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 will probably segfault calling the function. You also have to be careful when passing generic pointer values to external C routines, since currently there is no type checking for these; any pointer type other than `char*`, `expr*` and the matrix pointer types is effectively treated as `void*`. This considerably simplifies lowlevel programming and interfacing to C libraries, but also makes it very easy to call C functions in inappropriate ways. Therefore it is highly recommended that you wrap your lowlevel code in Pure routines and data structures which do all the checks necessary to ensure that only the right kind of data is passed to C routines.

Another limitation of the C interface is that it does not offer any special support for C structs and C function parameters right now. However, an optional addon module is available which uses `libffi`, a portable foreign function interface library, to provide that kind of functionality. This also makes it possible to turn Pure closures into C callback functions, which is needed for interfacing to some C libraries, without writing a single line of C code. Please see the description of the `pure-ffi` module on the *Addons* wiki page for details.

## 7.6 Compiling Scripts

While Pure is typically used in an interactive way, it is also possible to compile your scripts to native executables which can be run without the interpreter. Basically, all you have to do is to add the `-c` option when running your script with the interpreter, and to specify the desired output filename with the `-o` option:[4]

```
$ pure -c hello.pure -o hello
```

The given script is then executed as usual, but after execution the interpreter takes a snapshot of the program and compiles it to an executable. Alternatively, it is also possible to create either an LLVM assembler (`.ll`) or bitcode (`.bc`) file, or a native assembler (`.s`) or object (`.o`) file, depending on the output filename specified with `-o`. The `.ll` and `.bc` formats are supported natively by the Pure interpreter, no external tools

[3] See `http://code.google.com/p/pure-lang/wiki/Addons`.

[4] This requires the basic LLVM toolchain to be installed. See the Pure manual and the installation instructions for details.

are required to generate these. If the target is a `.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 `gcc` to assemble and link the resulting program (if requested). Below we concentrate on compiling executables, because it is the simplest and most common case. Please refer to the Pure manual for information on how to build object modules and link them into programs and libraries.

One advantage of compiling your script is that this eliminates the JIT compilation phase 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 Pure manual for details.

An unusual feature of Pure's batch compiler is that the compiled program is actually executed as usual, i.e., the script is run *at compile time*, too. This might first seem to be a big annoyance, but it actually opens the door for some powerful programming techniques like *partial evaluation*; we'll illustrate this below with a simple example. 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, and using the built-in `eval` function 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.

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
```

Turning the script into an executable works as follows:

```
$ pure -c hello.pure -o hello
$ ./hello 10
Hello, world!
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

That was easy. Now let's see how we can 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` is not specified on the command line.)

Moreover, we want to skip the execution of `main` at compile time. The Pure runtime provides a special system variable `compiling` which 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]
```

This technique is also known as *partial evaluation*. 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. Also note that the compile time parameters, like `n` in this example, can actually be *anything* that a Pure script can compute and represent as a compile time constant. This is possible because the script really gets executed at compile time, which is something that most other programming language compilers can't do.

Chapter 8

# Examples

This chapter shows how to tackle different kinds of typical programming tasks using Pure and illustrates a variety of important programming techniques. While we try to cover a few important areas, the selection is not exhaustive (which would be rather impossible, given the breadth of computer applications nowadays), and the scale of the problems discussed here is somewhat trimmed down to fit the scope of an introduction. But they are not just academic exercises either, and so we hope that you at least get a glance of how Pure can be employed as a practical problem solving tool.

## 8.1 Recursion

This section discusses recursion, one of the most important concepts in functional programming. As a concrete example, we take another look at our good old friend, the Fibonacci function. This is a fairly basic example, but still interesting enough to discuss various important techniques such as tabulation and tail recursion.

First, here's the naive definition of the Fibonacci function we already saw in Chapter 2:

```
> fib n = if n<=1 then n else fib (n-2) + fib (n-1);
> map fib (0..10);
[0,1,1,2,3,5,8,13,21,34,55]
```

This is a really bad implementation because it takes exponential running time, which becomes quite apparent if you try to compute `fib n` for larger values of `n`. In fact, you will find that the ratio between the running times of successive invocations quickly starts approaching the golden ratio $\varphi = 1.618\ldots$, which is no accident because the times are proportional to the Fibonacci function itself! So, even assuming a fast computer which can do each single call to `fib` in just a nanosecond, a conservative estimate of the time needed to compute just the 128th Fibonacci number would already exceed the current age of the universe by some 29.6%. But we can avoid this defect if we generate the Fibonacci numbers in pairs instead:

```
> fibs n = 0,1 if n<=0;
>        = b,a+b when a,b = fibs (n-1) end otherwise;
> fibs 10; fibs 30;
55,89
832040,1346269
```

Note the **when** clause (cf. Chapter 4), which extracts the individual results from the pair returned by the recursive invocation of `fibs` in the second equation. The underlying technique here is *tabulation*, i.e., keeping track of intermediate results in a table.

It's now an easy exercise to wrap the above definition in a `fib` function which just grabs the first result from a corresponding call to `fibs`. But we first have to get rid of the old definition of the `fib` function; we can do that in the interpreter as follows:

```
> clear fib
```

Now enter the following definition. Note that `fibs` becomes a local function inside `fib` now. This kind of "wrapper-worker" design is pretty common in functional programs.

```
> fib n = a when a,b = fibs n end with
>   fibs n = 0,1 if n<=0;
>          = b,a+b when a,b = fibs (n-1) end otherwise;
> end;
> map fib (21..31);
[10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269]
```

Our new definition is much better than the naive one, but it still has some deficiencies:

- The Fibonacci numbers outgrow the 32 bit range of machine integers pretty quickly and then start wrapping around. E.g., `fib 47` yields `-1323752223`.
- Our definition is in danger of running out of stack space because of the recursive invocations of the `fibs` function.

The first defect is easy to fix. In order to get arithmetically correct results, we just have to switch to bigints. In fact, it is enough to replace the starting values of the sequence to force the entire computation to be done using bigints (note that the `L` suffix tells the interpreter that these are now bigint values rather than ordinary machine integers):

```
> clear fib
> fib n = a when a,b = fibs n end with
>   fibs n = 0L,1L if n<=0;
>          = b,a+b when a,b = fibs (n-1) end otherwise;
> end;
> map fib (47..51);
```

```
[2971215073L,4807526976L,7778742049L,12586269025L,20365011074L]
> fib 200;
280571172992510140037611932413038677189525L
```

The second issue only becomes apparent when we want to compute some *really* huge Fibonacci numbers, much larger than the ones we tried so far. But first make sure that we save our work at this point so that we don't have to retype the definition of `fib` later:

```
> dump
```

This command writes our current definitions to a script file named `.pure` in the current directory, which will be reloaded automatically the next time we start the interpreter. After this safety measure let's give it a shot:

```
> fib 1000000;
Segmentation fault
```

Oops. (You did save your work with `dump`, didn't you?) By default, the Pure interpreter doesn't do any stack checks so it crashed with a stack overflow. At this point you should probably read up on "stack size and tail recursion" in the Pure manual and set the `PURE_STACK` environment variable to a sane value (cf. Section 7.2). For instance:

```
$ export PURE_STACK=4096
```

Let's restart the interpreter now and try again:

```
> fib 1000000;
<stdin>, line 1: unhandled exception 'stack_fault' while evaluating
'fib 1000000'
```

Ok, at least we get an orderly exception now, but of course that doesn't fix the problem. The solution here is to rewrite the definition of `fibs` so that it becomes tail-recursive, cf. Section 7.1. This is a very important concept in functional programming, because it enables us to implement iteration in limited stack space. Recall that a function definition is *tail-recursive* if the recursive invocation of the function is the last call (the so-called *tail call*) on the right-hand side of the definition. Tail calls are usually optimized away by the Pure compiler so that a tail-recursive function can execute in a loop-like fashion.

The easiest trick to turn a recursive function into a tail-recursive one is the *accumulating parameter* technique. The idea is to have our "worker" function carry around an extra argument representing the latest intermediate result of the iteration. For the Fibonacci function this is quite easy; we just take the current pair `a,b` of Fibonacci numbers as the accumulating parameter:

```
> clear fib
> fib n = fibs n (0L,1L) with
>   fibs n (a,b) = a if n<=0;
>                = fibs (n-1) (b,a+b) otherwise;
> end;
```

Let's take our final definition for a first test drive:

```
> map fib (0..10);
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L,55L]
```

Looks good so far. Now to "boldly go ..." (Sit back and relax, this takes a little while; the result has 208988 digits.)

```
> fib 1000000;
1953282128.......
```

## 8.2 Lists and Streams

## 8.3 Matrix Operations

## 8.4 String Processing

## 8.5 Sorting and Searching

## 8.6 Symbolic Computing

## 8.7 System Programming

## 8.8 Databases

## 8.9 Web Programming

## 8.10 Computer Graphics

## 8.11 Multimedia and Computer Music

Appendix A

# Pure Grammar

This is the complete extended BNF grammar of Pure. As usual, repetitions and optional elements are denoted using curly braces and brackets, respectively. For the sake of simplicity, the grammar leaves the precedence and associativity of expressions unspecified; you can find these in Chapter 4.

*script* : {*item*}

*item* : **namespace** [*name*] ;
| **using namespace** [*name* {, *name*}] ;
| **using** *name* {, *name*};
| [*scope*] **extern** *prototype* {, *prototype*};
| *declarator symbol* {*symbol*};
| **let** *simple-rule*;
| **const** *simple-rule*;
| **def** *simple-rule*;
| *rule*;
| *expr*;
| ;

*declarator* : *scope* | [*scope*] *fixity*

*scope* : **public** | **private**

*fixity* : **nonfix** | **outfix**
| **infix** *precedence* | **infixl** *precedence* | **infixr** *precedence*

```
                | prefix precedence | postfix precedence

precedence      : integer | ( qualified-symbol )

prototype       : c-type identifier ( [parameters] ) [= identifier]

parameters      : parameter {, parameter}

parameter       : c-type [identifier]

c-type          : identifier {*}

symbol          : identifier | special

name            : qualified-identifier | string

rule            : pattern {| pattern} = expr [guard] {; = expr [guard] }

simple-rule     : pattern = expr | expr

pattern         : simple-expr

guard           : if simple-expr
                | otherwise
                | guard when simple-rules end
                | guard with rules end

expr            : \ prim-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 qualified-symbol simple-expr
```

| | |
|---|---|
| | \| *qualified-symbol simple-expr* |
| | \| *simple-expr qualified-symbol* |
| | \| *application* |
| *application* | : *application prim-expr* |
| | \| *prim-expr* |
| *prim-expr* | : *qualified-identifier* [`::` *qualified-identifier* \| `@` *prim-expr*] |
| | \| *qualified-symbol* |
| | \| *number* |
| | \| *string* |
| | \| `(` *qualified-symbol* `)` |
| | \| `(` *qualified-symbol qualified-symbol* `)` |
| | \| `(` *simple-expr qualified-symbol* `)` |
| | \| `(` *qualified-symbol simple-expr* `)` |
| | \| `(` *expr* `)` |
| | \| *qualified-symbol expr qualified-symbol* |
| | \| `[` *exprs* `]` |
| | \| `{` *exprs* {`;` *exprs*} [`;`] `}` |
| | \| `[` *expr* `\|` *simple-rules* `]` |
| | \| `{` *expr* `\|` *simple-rules* `}` |
| *exprs* | : *expr* {`,` *expr*} |
| *rules* | : *rule* {`;` *rule*} [`;`] |
| *simple-rules* | : *simple-rule* {`;` *simple-rule*} [`;`] |
| *qualified-symbol* | : [*qualifier*] *symbol* |
| *qualified-identifier* | : [*qualifier*] *identifier* |
| *qualifier* | : [*identifier*] `::` {*identifier* `::`} |
| *number* | : *integer* \| *integer* `L` \| *float* |

*integer* : *digit* {*digit*}
| `0` (`X`|`x`) *hex-digit* {*hex-digit*}
| `0` (`B`|`b`) *bin-digit* {*bin-digit*}
| `0` *oct-digit* {*oct-digit*}

*float* : *digit* {*digit*} [`.` *digit* {*digit*}] *exponent*
| [*digit* {*digit*}] `.` *digit* {*digit*} [*exponent*]

*exponent* : (`E`|`e`) [`+`|`-`] *digit* {*digit*}

*string* : `"` {*char*} `"`

*identifier* : *letter* {*letter* | *digit*}

*special* : *punct* {*punct*}

*digit* : `0`|···|`9`

*oct-digit* : `0`|···|`7`

*hex-digit* : `0`|···|`9`|`A`|···|`F`|`a`|···|`f`

*bin-digit* : `0`|`1`

*letter* : `A`|···|`Z`|`a`|···|`z`|`_`|···

*punct* : `!`|`#`|`$`|`%`|`&`|···

*char* : ⟨any character or escape sequence⟩

Note that the character repertoire available for the lexical entities *letter*, *punct* and *char* depends on the basic character set that you use. Pure supports either just 7 bit ASCII, or all of Unicode. In the latter case, your scripts must be encoded in UTF-8, an ASCII extension which can represent all extended Unicode characters using special multibyte sequences. Most text editors support UTF-8 nowadays, so that you can write

your Pure programs in almost any language, and take full advantage of the special symbols in the Unicode character set.

A string character can be any character in the host character set, except newline, double quote, the backslash and the null character (ASCII code 0, which, like in C, is used as a string terminator). As usual, the backslash is used to denote special escape sequences. In particular, the newline, double quote and backslash characters can be denoted `\n`, `\"` and `\\`, respectively. Pure provides escape sequences for all Unicode characters, which lets you use the full Unicode set in strings even if your editor only supports ASCII. Please see Chapter 3 for details.

Concerning identifiers and special symbols, Pure uses the following simplified rules (suggested by John Cowan on the Pure mailing list) to determine which Unicode characters are letters and punctuation:

- In addition to the ASCII punctuation symbols, Pure considers the following extended Unicode characters (code points) as punctuation which can be used in special operator and constant symbols: `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 following additional blocks of Unicode symbols: 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 blocks. This should cover almost everything you'd ever want to use in operator symbols.

- All other extended Unicode characters are considered as letters which can be used in ordinary identifiers just as well as the alphabetic characters in the ASCII set (and the underscore).

# Bibliography


[1] F. Baader and T. Nipkow. *Term Rewriting and all that*. Cambridge University Press, Cambridge, 1998.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[3] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier, 1990.

[4] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 228–244. Springer, 1994.

[5] J. W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002. See `http://www.octave.org`.

[6] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[7] A. Gräf. Left-to-right tree pattern matching. In R. V. Book, editor, *Rewriting Techniques and Applications*, LNCS 488, pages 323–334. Springer, 1991.

[8] A. Gräf. *The Q Programming Language*. `http://q-lang.sf.net`, 2008.

[9] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[10] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. `http://haskell.org`, September 2002.

[11] C. Lattner et al. The LLVM compiler infrastructure. `http://llvm.org`, 2008.

[12] W. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley, 1988.

[13] J. N. Little and C. B. Moler. *MATLAB User's Guide*. MathWorks, Inc., Cochinate Place, 24 Prime Park Way, Natick, MA 01760, Jan. 1990.

[14] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006. See `http://www.ps.uni-sb.de/alice`.

[15] M. O'Donnell. *Equational Logic as a Programming Language*. Series in the Foundations of Computing. MIT Press, Cambridge, Mass., 1985.

[16] D. A. Turner. An overview of Miranda. In D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 1–16. 1990. See `http://miranda.org.uk`.

[17] W. van Oortmerssen. *Concurrent Tree Space Transformation in the Aardappel Programming Language*. PhD thesis, University of Southampton, UK, 2000.

[18] A. N. Whitehead. *A Treatise on Universal Algebra, with Applications*. Cambridge, 1898. Reprinted 1960.

[19] Wikipedia article "Functional programming". `http://en.wikipedia.org/wiki/Functional_programming`, October 2008.