Pure Quick Reference

Albert Gräf

April 7, 2018

Abstract

This is a quick reference guide to the Pure programming language for the impatient. It briefly summarizes all important language constructs and gives a few basic examples, so that seasoned programmers can pick up the language at a glance and start hacking away as quickly as possible.

Contents

1 Introduction

Pure is a functional programming language based on term rewriting. Thus your programs are essentially just collections of symbolic equations which the interpreter uses to reduce expressions to their simplest ("normal") form. Term rewriting makes for a simple but powerful and flexible programming model featuring dynamic typing and general polymorphism. In addition, Pure programs are compiled to efficient native code on the fly, using the LLVM compiler framework, so programs are executed reasonably fast and interfacing to C is easy.

On the surface, Pure looks similar to modern-style functional languages of the Miranda and Haskell variety, but under the hood it is a much more dynamic language, with macros and reflective capabilities more akin to Lisp. Pure's algebraic programming style probably appeals most to mathematically inclined programmers, but its interactive programming environment and easy extensibility also make it usable as a (compiled) scripting language for various application areas, such as graphics, multimedia, scientific, system and web programming. While languages like Haskell and ML cover much of the same ground, we think that Pure's feature set is different enough (and even unique in some ways) to make it an interesting alternative.

Like all programming languages, Pure also has its weak points. In particular, the lack of static typing, while appreciated by dynamic language aficionados, also means less type safety and more execution overhead, so Pure isn't the best language for large projects or heavy-duty number crunching. On the other hand it's a great little language to get your feet wet with modern functional programming and explore the symbolic capabilities of term rewriting, and the library support is certainly good enough for practical programming purposes as well.

1.1 Background and Recommended Reading

It will be helpful if you already have at least a passing familiarity with functional programming, see, e.g., [11], or [22] 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 [5]; we also

give a brief summary of the relevant notions in Appendix B. Term rewriting as a programming language was pioneered by Michael O'Donnell [18], 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 [8] and OPAL [6].

Pure is most closely related to its predecessor Q [10] and Wouter van Oortmerssen's Aardappel [21], although quite obviously it also heavily borrows ideas from other modern functional languages, in particular Miranda [20], Haskell [13] and Alice ML [17]. Pure's outfix operators were adopted from William Leler's Bertrand language [15], while its matrix support was inspired by MATLAB [16] and GNU Octave [7]. The pattern matching algorithm, which is the main workhorse behind Pure's term rewriting machinery, is described in [9]. To our knowledge, this is still basically the fastest known general left-to-right term matching technique, although some improvements have been reported in [19].

Pure also relies on other open source software, most notably the compiler framework LLVM [14] which Pure uses as its backend for doing JIT compilation, as well as the GNU Multiprecision Library for its bigint support.

1.2 Getting Started

The Pure interpreter is available at `https://agraef.github.io/pure-lang/`. There you can also find a mailing list and a wiki which has information to help you get up and running quickly. The documentation can be found online in a collection of manuals called the *Pure Language and Library Documentation*, which covers the Pure language and standard library as well as all the other addon modules and libraries available from the Pure website. The online documentation can also be read with the `help` command of the interpreter.

To run the interpreter, simply type the command '`pure`' at the shell command line. The interpreter then prints its sign-on message and leaves you at its '`> `' command prompt, where you can start typing definitions and expressions to be evaluated:

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

Typing `quit` or the end-of-file character at the beginning of the command line exits the interpreter and takes you back to the shell. The interpreter understands a number of other special interactive commands, see Section 7.2 for a complete list of these. In particular, we'll frequently use the `show` command to print the definitions of defined functions and variables:

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

Program examples are set in typewriter font; keywords of the Pure language are in boldface. These code snippets 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 '`> `', as in the samples above, this indicates an example interaction with the interpreter. Everything following the prompt is meant to be typed exactly as written. Lines lacking the '`> `' prefix show results printed by the interpreter.

More information about using the interpreter can be found in Section 7, and a few sample Pure programs have been included for your perusal, see Section 8. The other sections of this guide describe the important constructs and features of the Pure language. For reference purposes, the EBNF grammar of the language is listed in Appendix A, and Appendix B gives a brief account of the term rewriting theory underlying Pure's model of computation.

2 Lexical Matters

Pure is a *free-format* language, i.e., whitespace is insignificant (unless it is used to delimit other symbols). Thus, in contrast to layout-based languages like Haskell and Python, you *must* use the proper delimiters ('`;`') and keywords (**`end`**) to terminate definitions and block structures. This is also true for interactive usage; the interpreter basically accepts the same input language.

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.

Operators and *constant symbols* are special symbols which *must* be declared before they can be used, as explained in Section 3.3. 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 case 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 symbols.)

The Pure language also has some keywords which cannot be used as identifiers; these are listed in Appendix A. 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.

Pure fully supports the *Unicode* character set or, more precisely, UTF-8. This is an ASCII extension capable of representing all Unicode characters, which provides you with thousands of characters from most of the languages of the world, as well as an abundance of special symbols for almost any purpose. If your text editor supports the UTF-8 encoding (most editors do nowadays), you can use all Unicode characters in your Pure programs, not only inside strings, but also for denoting identifiers and special (operator 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 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 blocks of local function and variable definitions. The different kinds of expressions understood by the Pure interpreter are summarized in the following table, in order of increasing precedence.

Type	Examples	Description
Block	`\x y->2*x-y`	anonymous function (lambda)
	`case f u of x,y = x+y end`	pattern-matching conditional
	`x+y when x,y = f u end`	local variable definition
	`f u with f (x,y) = x+y end`	local function definition
Conditional	`if x>0 then x else -x`	conditional expression
Simple	`x+y`, `-x`, `x mod y`, `not x`	operator application
	`sin x`, `max a b`	function application
Primary	`4711`, `4711L`, `1.2e-3`	number
	`"Hello, world!\n"`	string
	`foo`, `x`, `(+)`	function or variable symbol
	`[1,2,3]`, `{1,2;3,4}`, `(1,2,3)`	list, matrix, tuple
	`[x,-y | x=1..n; y=1..m; x<y]`	list comprehension
	`{i==j | i=1..n; j=1..m}`	matrix comprehension

3.1 Primary Expressions

Primary expressions are the basic building blocks of expressions. Pure provides the usual C-like notations for identifiers, integers, floating point numbers and strings (see Section 2), as well as some special constructs to denote compound primaries (lists, tuples, matrices and records).

- *Symbols* come in two kinds: *Identifiers* are the usual sequences of letters (including the underscore) and digits, starting with a letter. These are used to denote functions and variables. *Special symbols* are used to denote operators and constant symbols; these may also consist of punctuation and must be declared explicitly (cf. Section 3.3).

- *Integers* can be denoted in decimal (`1000`), hexadecimal (`0x3e8`), octal (`01750`) and binary (`0b1111101000`). By default, these denote 32 bit signed machine integers. *Bigints* (arbitrary precision integers, which are implemented using the GNU Multiprecision Library) are indicated with the suffix `L`, e.g., `1000L`. Also, large integer constants exceeding the 32 bit range are promoted to bigints automatically.

- *Floating point numbers* are always denoted in decimal. To distinguish these from integers, they must always contain a decimal point and/or a scale factor (power of 10 exponent), as in `1.2e-3`. Internally, these are always stored with double precision (64 bit).

- *Strings* are arbitrary character sequences enclosed in double quotes, such as `"abc"` or `"Hello, world!\n"`. These are always encoded in UTF-8 internally.

- *Lists* are written using brackets, such as `[1,2,3]`. These are in fact just syntactic sugar for the `':'` operator (cf. Section 3.4), thus `[1,2,3]` is exactly the same as `1:2:3:[]`, where `[]` denotes the empty list. Lists may be nested, and may be polymorphic (contain elements of different types), such as `[1,[2,3],foo 5 6]`.

- *Tuples* are a "flat" kind of 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 `[]`, but have the following additional properties:

  – 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 this implies that tuples cannot 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). Tuples thus provide a convenient way to represent plain sequences which don't need an elaborate, hierarchical structure.

- *Matrices* are written using curly braces, using the semicolon to separate different rows. These work just like in Octave or MATLAB. `{1,2,3}` denotes a row vector (1×3 matrix), `{1;2;3}` a column vector (3×1 matrix), and `{1,2,3;4,5,6}` a 2×3 matrix. 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;4},{2;5},{3;6}}` is another way to write the 2×3 matrix `{1,2,3;4,5,6}` in "column-major" form. *Numeric matrices* use an internal representation which is compatible with the GNU Scientific Library; they must be homogeneous and contain either integer, floating point or complex values only. Pure also supports *symbolic matrices* which may contain any mixture of Pure expressions, such as `{1,[2,3],foo 5 6}`. The *empty matrix* is denoted `{}`; this is by default a symbolic 0×0 matrix.

- *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.

- *Comprehensions* provide a means to construct lists and matrix values using a variation of mathematical set notation, by drawing elements from other lists and matrices (*generator clauses*), possibly restricting the range of collected elements using predicates (*filter clauses*). For instance, `[2*x|x=xs;x>0]` denotes the list of all `2*x` for which `x` runs through all the positive elements of the list (or matrix) `xs`. See Section 3.9 below for details.

3.2 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 $f\ x\ y = (f\ x)\ y$. This style of writing function applications is also known as *currying*, after the American logician Haskell B. Curry who popularized its use through his work on the combinatorial calculus.

Currying is much more than just a notational convenience; it's a way to transform applications of a function to multiple arguments into a series of single-argument applications. Specifically, if f is a function taking two arguments x and y, then $f\ x$ becomes a function in its own right, namely the function which maps each given y to $f\ x\ y$. Currying thus makes it possible to derive new functions from existing ones by just omitting trailing arguments, which yields a so-called *unsaturated* or *partial application*. Conversely, an application of a function which supplies all needed parameters and is thus "ready to go" is called *saturated*.

Taking the prelude function `max` as an example, the partial application `max 0` thus denotes a function which returns its argument x if it's positive, and zero otherwise (mathematicians also call this the *positive part* x^+ of x). So we may write:

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

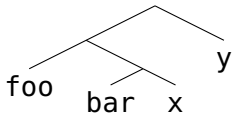
Function applications are normally evaluated from left to right in Pure, innermost expressions first. This is also known as *applicative order* or *call-by-value*, since the arguments of a function (as well as the function object itself) are evaluated before the function is applied. For instance, consider the following function `square`:

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

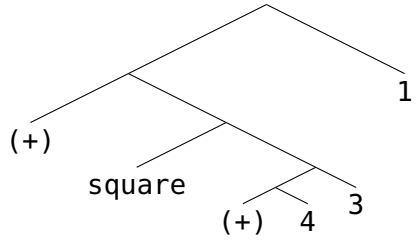
Using that definition as well as the built-in rules of arithmetic, the evaluation of the expression `square (4+3)+1` proceeds as follows. In particular, note that the reducible subterm `4+3` supplied as an argument to `square` gets evaluated first (such a reducible expression is also called a *redex* in term rewriting parlance, and the term it reduces to is called a *reduct*).

```
square (4+3)+1 = square 7+1 = 7*7+1 = 49+1 = 50
```

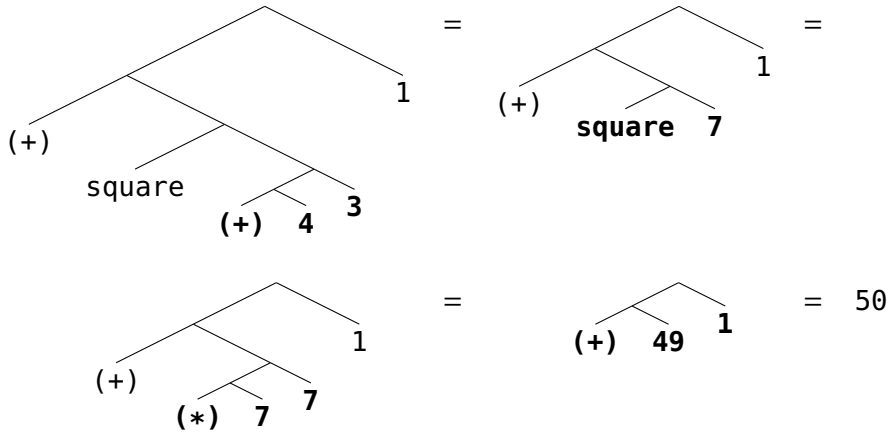
It is often helpful to depict an expression as a binary tree which has the curried function applications as interior nodes and the functions, constants and variables at the leaves of the tree. For instance, `foo (bar x) y` may be depicted as follows (as usual, we draw the tree like the computer scientists do, i.e., upside-down, with the root of the tree at the top):



Expressions involving operators can be visualized in the same manner. As we'll see in Section 3.3, an infix expression like `x+y` is really just an application `(+) x y` of the '+' function, denoted `(+)`. So the expression `square (4+3)+1` can be depicted as:



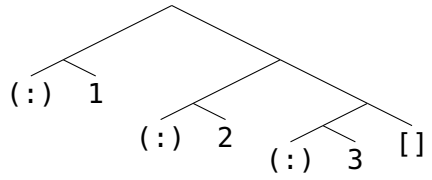
With this visualization aid, we may consider the evaluation of `square (4+3)+1` as a series of *tree transformations*, which in each step replaces a subtree, the redex, with another subtree, the reduct. In the following picture, the redices are denoted in boldface:



Here we got an atomic value, a number, as the resulting normal form. This isn't always the case, however. Sometimes even a saturated function application may not be evaluated at all, since there is no applicable definition.[1] In such cases the application itself is returned as a normal form expression, and the applied function becomes a

[1] In fact, the function operand of an application doesn't even have to be a "function" in Pure. E.g., it might be a number, as in `5 2`. Such terms are always irreducible in Pure.

constructor. This is the case, in particular, for the list constructor `':'` declared in the prelude. Thus a list like `[1,2,3]` (which, as we've learned, is just a shorthand for `1:2:3:[]`) corresponds to the following expression tree:



In this case the constructor symbol `':'` is a *pure* constructor, i.e., there are no defining equations for `':'` at all. But literal applications are also constructed if the applied function *is* defined, yet there's no rule which covers the specific case at hand. By these means, *any* function symbol may become part of a normal form expression:

```
> map (+1) [a,b,c];
[a+1,b+1,c+1]
```

This is probably one of Pure's most unusual aspects and may need some getting used to. But this is just how term rewriting works, and it's also what makes symbolic evaluations possible in Pure. Also, as the list example shows, constructor applications let us represent any kind of hierarchical data structure in an algebraic way. Section 4.3 explains how to define functions operating on such values.

3.3 Operators

For convenience, Pure also lets you introduce special constant and operator symbols using a so-called *fixity* declaration. This is nothing but syntactic sugar; internally, an operator application is actually represented as a curried function application. You can also turn an operator into an ordinary function symbol by enclosing it in parentheses. E.g., `x+y` is in fact exactly the same as `(+) x y`, just using a somewhat prettier and more familiar notation.

Fixity declarations take 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 a whitespace-delimited list of symbols (identifiers or punctuation). At each level, non-associative operators have the lowest precedence, followed by left-associative, right-associative, prefix and postfix symbols.

For the infix operators, Pure provides the usual Haskell/Miranda-style *operator sections* of the form `(`*x*`+)` (left section) or `(+`*x*`)` (right section) as a shorthand for partial operator applications. 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 `(-`*x*`)` is always interpreted as an instance of unary minus; a function which subtracts *x* from its argument can be written as `(+-`*x*`)`.)

In addition to these fairly common kinds of operators, Pure also has outfix and nonfix symbols. *Outfix operators* 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)`.

Nonfix symbols are nullary operators, 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 3.5). For instance:

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

All the special kinds of symbols discussed above effectively become keywords of the language and cannot be used as ordinary function or variable identifiers any more. (In contrast to ordinary keywords, however, they may still be qualified with a namespace identifier, see Section 5.2. Thus the status of such a symbol actually depends on which namespaces are in scope at a given point in the program.)

[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.

3.4 Predefined Operators

The following operators are predefined in the prelude. 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  1100   $ ;                // right-associative application
infixr  1200   , ;                // pair (tuple)
infix   1300   => ;               // key=>value pairs ("hash rocket")
infix   1400   .. ;               // arithmetic sequences
infixr  1500   || ;               // logical or (short-circuit)
infixr  1600   && ;               // logical and (short-circuit)
prefix  1700   ~ ;                // logical negation
infix   1800   < > <= >= == ~= ;  // relations
infix   1800   === ~== ;          // syntactic equality
infixr  1900   : ;                // list cons
infix   2000   +: <: ;            // complex numbers (cf. math.pure)
infixl  2100   << >> ;            // bit shifts
infixl  2200   + - or ;           // addition, bitwise or
infixl  2300   * / div mod and ;  // multiplication, bitwise and
infixl  2300   % ;                // exact division (cf. math.pure)
prefix  2400   not ;              // bitwise not
infixr  2500   ^ ;                // exponentiation
prefix  2600   # ;                // size operator
infixl  2700   ! !! ;             // indexing, slicing
infixr  2800   . ;                // function composition
prefix  2900   ' ;                // quote
postfix 3000   & ;                // thunk
```

Here is a brief description of the basic arithmetic and logical operations:

- `-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. (If the `math` module is not loaded then '`%`' acts as a simple constructor symbol.)

- `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 defined in the prelude are either function combinators, specialized data constructors or operations to deal with lists and other aggregate structures:

- `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*. E.g., `1..5` evaluates to `[1,2,3,4,5]`. `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 simple aggregate values in Pure. As already mentioned, the pair constructor is associative in Pure, so that, in contrast 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.)

[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`.

- `x!y`: The *indexing* operation. This somewhat peculiar notation seems to have its origins in the BCPL language. (Pure inherited it from Q which in turn adopted it from the original edition of the Bird/Wadler book [3].) 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 `%`, these require the `math` module to work (they will act as simple constructors without defining equations if the `math` module is not loaded). The "hash rocket" `=>` is a plain constructor symbol which is used to denote key-value associations.

- `x$$y`, `x&`, `'x`: These operators are special forms, cf. Section 3.10. `x$$y` is used to execute expressions in sequence, `x&` creates "thunks" a.k.a. "futures" which are evaluated lazily, and `'x` (or, equivalently, `quote x`) defers the evaluation of an expression.

3.5 Patterns

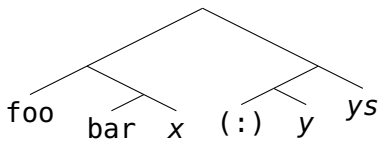
Patterns are pervasive in Pure. They form the left-hand sides of rules in all function definitions and variable-binding constructs to be discussed in Sections 3.7 and 4. Patterns are expressions which serve as templates to be *matched* against a subject term. If the subject matches the literal parts of a pattern, variables in the pattern are bound to the corresponding values in the subject.

In the simplest case, a pattern may just be a lone variable, but in the general case it can be any (simple) expression. In Pure, a *variable* is any identifier at the "leaves" (atomic subexpressions) of the pattern, subject to the following constraints:

- *Head = function* (read: "head is function") rule: Identifiers in the head position of a function application always denote literal function symbols.
- *Nonfix symbols* (cf. Section 3.3) are always interpreted as literals and can never be variables.

Pure is a terse language. The "head = function" rule is simply a convention which lets us get away without declaring the variables or imposing some special (usually awkward) lexical syntax. Constant symbols need to be declared as nonfix, since otherwise they couldn't be distinguished from variables; but these are quite rare, at least compared to variables in patterns which are a dime a dozen.

To explain the "head = function" rule, let's consider the pattern `foo (bar x) (y:ys)` as an example. You can see at a glance (at least with some practice) that `foo` and `bar` as well as the list constructor `(:)` are the function symbols, whereas `x`, `y` and `ys` are the variables. Note that `(:)` is indeed the head symbol of the application `y:ys` here, because in curried application syntax this expression is written as `(:) y ys`. To better see this, let's depict the expression as a binary tree, in the same way as in Section 3.2:



The variables have been marked with italics here; they are the leaves dangling from the right branches of the tree (also called the *variable positions*), while the function symbols in *non-variable* or *head positions* can all be found at the left branches. Note that only *identifiers* at variable positions can be variables; constants like numbers will of course always be interpreted as literals, no matter where they are located in the expression tree. The same is true for nonfix symbols, which are always interpreted as literal constants, as far as pattern-matching is concerned.

The variable `_` is special in patterns. It denotes the *anonymous variable*, which matches an arbitrary value (independently for all occurrences) and does not bind a variable value. For instance, `_:1:_` matches any list with the integer 1 in the second element. Also note that the anonymous variable is exempt from the "head = function" rule, so it is always interpreted as a variable, even if it occurs in the head position of a function application.

Patterns may be *non-linear* in Pure, i.e., they may contain multiple occurrences of a variable. All occurrences of the same variable (other than the anonymous variable) must then 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 []       = [];
```

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. (Pure clearly distinguishes this from the *semantic equality* predicate embodied by the '==' and '~=' operators which can be defined freely by the programmer.) Syntactic equality is also available as an explicit operation `same` as well as corresponding operators '===' and '~==' in the prelude, so that the first rule above is roughly equivalent to:

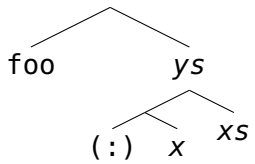
```
uniq (x:y:xs) = uniq (x:xs) if x === y;
```

Syntactically, patterns are just simple expressions, but they may also contain the following special elements which are not permitted in ordinary expressions:

- *"As" pattern:* This is a pattern of the form *variable*`@`*pattern* which, in addition to the given variable, also matches the given subpattern and binds the variables in the subpattern accordingly. Syntactically, "as" patterns are primary expressions; if the subpattern is not a primary expression, it must be parenthesized. For instance, `xs@(x:_)` matches a non-empty list and binds the variable `xs` to the entire list and the variable `x` to its head element.

- *Type tags:* A variable can be followed by the `::` symbol and a type symbol, to indicate that it can only match a value of the corresponding type. There are a few built-in types, namely `int`, `bigint`, `double`, `string`, `matrix` and `pointer`, which denote the corresponding primitive types built into the language. Some other types are defined in the prelude, and additional types can be defined by the programmer as needed.

"As" patterns place variables at the interior (non-leaf) nodes of an expression tree. E.g., `foo (ys@(x:xs))` may be depicted as:



Note that the "as" pattern `x@_` matches any value anywhere and binds the variable `x` to it. This works just like `x` itself, except in non-variable (i.e., head) positions where a lone `x` would be interpreted as a literal. Thus a pattern like `x@_ y` provides a means to "escape" a variable in the head position of a function application. This is handy if you need to define functions operating on function applications in a generic way. For instance, the following little example illustrates how you can collect the function and arguments of an application in a list:

```
> appl x = a [] x with a xs (x@_ y) = a (y:xs) x; a xs x = x:xs end;
> appl (f x y z);
[f,x,y,z]
```

Type tags are a kind of additional "guards" on a definition, which restrict the set of terms that can be matched by the corresponding variable. Except for the built-in types, they must be introduced by means of a *type definition*. The format of these definitions is explained in Section 4.6. For the sake of a simple example, let us consider points in the plane which might be represented using a constructor symbol `Point` which gets applied to pairs of coordinates. We also 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:

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

3.6 Conditional Expressions

An expression of the form **`if`** x **`then`** y **`else`** z evaluates to y if x is a nonzero integer, or to z if x is zero. This is a special form which only evaluates one of the branches y and z, depending on the value of x. An exception is raised if x is not a machine integer. Example (the factorial):

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

Conditional expressions can be nested in the usual way to obtain multiway conditionals. This also includes the customary "**`else if`**" chains. For instance, here's one (rather inefficient) way to define the Fibonacci function, which computes the Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, ...:

```
fib n = if n==0 then 0 else if n==1 then 1 else fib (n-2) + fib (n-1);
```

3.7 Block Expressions

A number of special constructs are provided to define local functions, and to match expressions against a pattern and bind the variables in the pattern accordingly. A subexpression is then evaluated in the context of these local definitions. Note that the "*lhs* = *rhs*" rule format employed in these constructs works in the same fashion as in global definitions; we'll discuss this syntax in more detail in Section 4.2.

- \$x_1 \cdots x_n$ -> y, where $x_1, \ldots, x_n$ are primary expressions ($n \geq 1$), denotes a *lambda* function which maps the given patterns $x_1, \ldots, x_n$ to an expression y; the latter is also called the *body* of the lambda expression. The lambda expression returns an anonymous function which, when applied to n argument values $z_1, \ldots, z_n$, simultaneously matches all z_i against the corresponding x_i and returns the value of the body y after substituting the pattern variables. An exception is raised if the arguments z_i do not match the patterns x_i.

- **case** x **of** y_1 = z_1; y_2 = z_2; ... **end** is a multiway conditional which matches the value of x against each pattern y_i and, as soon as a pattern matches, returns the corresponding value z_i (after substituting the pattern variables) as the value of the **case** expression. An exception is raised if x doesn't match any of the patterns y_i.

- x **when** y_1 = z_1; y_2 = z_2; ... **end** does local variable bindings. Each pattern y_i is matched against the value of z_i, and finally x is evaluated in the context of the resulting variable bindings. The bindings are executed in sequence, so that each z_i can refer to all variables bound in previous rules y_j = z_j, $j = 1, \ldots, i-1$. An exception is raised if any of the y_i fails to match the value of the corresponding z_i.

- x **with** f y_1 y_2 ... = z; ... **end** defines local functions. These work like global function definitions (cf. Section 4) except that they have local scope and have access to all local functions and variables in their scope. All functions in a **with** clause are defined simultaneously, thus the definitions may be mutually recursive.

The first three forms are in fact all reducible to the **with** construct, using the following equivalences. (In these 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.)

\$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**	$\equiv$	**case** z **of** y = x **end**

Here are some examples to illustrate how these constructs work (more examples can be found in Section 4.2 and throughout this manual). Lambdas are typically used for little "one-off" functions passed as arguments to other functions, e.g.:

```
squares n = map (\x -> x*x) (1..n);
```

Using a local function, we might also write this more verbosely as:

```
squares n = map square (1..n) with square x = x*x end;
```

So lambdas are often more concise, but local functions are a lot more versatile; they can be recursive and consist of several rules which makes writing complicated definitions more convenient. For instance, here is a definition of the Fibonacci function which also illustrates the use of local variables bound with the **when** construct:

```
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;
```

We mention in passing that the algorithm we use here is much more efficient than our earlier naive definition of the Fibonacci function, since the local `fibs` function computes the Fibonacci numbers in pairs, which can be done in linear time. (This kind of "wrapper/worker" design is fairly common in functional languages. An even better algorithm along these lines is given in Section 8.2.)

Since the syntax of the **with** and **when** constructs looks quite similar, it is important to note the differences between the two. Both constructs consist of a target expression to be evaluated and a collection of rules. However, a **with** clause contains proper *rewriting rules* defining one or more functions; these definitions are all done simultaneously and may thus be mutually recursive. In contrast, a **when** clause consists of so-called *pattern bindings* which simply evaluate some expressions and match them against the left-hand side patterns in order to bind some local variables; these definitions are *not* recursive, rather they are executed in sequence, so that each definition may refer to variables defined earlier. So we may write, e.g.:

```
> 2*x when x = 99; x = x+2 end;
202
```

This looks a lot like imperative code, but it's purely functional: First `x` is bound to `99`, and then this value is used in the second definition to bind a *new* variable `x` to `x+2 = 101`, which then becomes the value of `x` used in the target expression `2*x` of the **when** clause. This "sequential execution" aspect is rather important, because it enables you to do a series of "actions" (variable bindings and expression evaluations) in sequence by simply enclosing it in a **when** clause; we'll discuss some further examples of this in Section 4.2.

As a mnemonic that helps to keep **with** and **when** apart, think of **when** as conveying a sense of time, indicating that its clauses are executed sequentially.

The **case** construct is similar to **when** in that it also binds local variables and then evaluates a target expression in that context. In fact, the equivalences stated above tell us that a **case** clause with a single rule works exactly like a **when** clause consisting of a single definition; the only difference is that here the expression to be matched comes first and the target expression last. Thus we might rewrite our earlier definition of the Fibonacci function as follows:

```
fib n = case fibs n of a,b = a end with
  fibs n = 0,1 if n<=0;
         = case fibs (n-1) of a,b = b,a+b end;
end;
```

However, more typically **case** expressions are employed when there really are multiple cases to consider; a default case may be indicated with the pattern _ (the anonymous variable) which matches everything. For instance, here's the factorial again, this time using **case**:

```
fact n = case n of 0 = 1; _ = n*fact (n-1) if n>0 end;
```

By combining these constructs in different ways you can direct the flow of computation and compose complicated functions with ease in Pure. So in a sense the constructs discussed here, along with recursion and conditional expressions, are Pure's "control structures". Choosing the "right" construct is often a matter of convenience and personal style, although there are some idiomatic uses which we've mostly covered above; more elaborate examples can be found in Section 8.

3.8 Lexical Scoping

The block expressions introduce a hierarchy of *local scopes* of identifiers, pretty much like local function and variable definitions in Algol-like block-structured languages. 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 is unbound in this example. 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
```

Local functions are handled in an analogous fashion, but there's another subtlety involved here, the so-called "funarg problem". Note that local functions are first-class objects in Pure which can be passed around just like any other value, and such a local function may refer to other local functions and variables in its own context. This isn't much of a problem if a local function is only passed "downwards" (as a function argument), since in this case all local entities a function refers to are still on the execution stack and thus readily available. But this isn't true anymore if a function is passed "upwards" (as a function result). In such a case lexical scoping dictates that the local

function object carries with it the bindings of *all* local entities it references, so that these live on and are accessible when the function later gets invoked in a different context.

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
```

As a lexical closure, the instance of `add` returned by `adder 5` thus has an invisible binding of the local `x` parameter to the value `5` attached to it. As demanded by lexical scoping, this works the same no matter what other global or local bindings of `x` may be in effect when our instance of `adder` is invoked:

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

3.9 Comprehensions

List and matrix *comprehensions* take the form `[` *x* `|` *clause*`; ...]` and `{` *x* `|` *clause*`; ...}`, respectively, where *x* is an arbitrary *template expression* and each *clause* can be either a *generator* or a *filter* clause.

- A *generator clause* takes the form *y* `=` *z* where *y* is a pattern to be matched against each member of the value of *z*, binding the variables in *y* accordingly; only those elements will be collected which match the pattern, other elements are quietly discarded. *z* must evaluate to a list or a matrix.[4]
- A *filter clause* is an expression *p* yielding a machine int; only those elements will be collected where the filter expression returns nonzero. An exception is raised if *p* produces anything else but a machine int.

The clauses are processed from left to right, and each clause may refer to all variables bound in earlier clauses. Finally the template expression is evaluated for each combination of bound variables, and the list or matrix of all resulting values becomes the value of the comprehension.

Comprehensions are really just syntactic sugar for combinations of lambdas, conditional expressions and various list and matrix operations. The interpreter does the

[4]Strings are permitted, too, and will be promoted to the corresponding list of characters. In fact, comprehensions can draw values from any kind of container structure which implements the necessary interface operations such as `catmap`.

necessary expansions at compile time. For instance, list comprehensions are essentially implemented according to the following equivalences:[5]

`[` 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:

```
> 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);
> foo 3 4;
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

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, so that customary mathematical notation works as expected:

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

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

```
> a::number*xs::matrix = {a*x|x=xs};
> 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 or nontrivial patterns), otherwise an exception is raised at runtime.

3.10 Special Forms

Pure normally evaluates expressions using *call-by-value*, i.e., all subexpressions of an expression are evaluated before the expression itself. However, as already mentioned, some operations are actually implemented as *special forms* which defer the evaluation of some or all of their arguments until they are actually needed (i.e., doing *call-by-name* evaluation). The most important predefined special forms are listed below.

[5] These rules assume that y is an unqualified variable. The case of a nontrivial pattern y is handled in a fashion similar to filter clauses, in order to filter out unmatched elements in generator clauses.

- 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*.

- The logical connectives `&&` and `||` evaluate their operands in short-circuit mode. 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 `$$` evaluates its left operand, immediately throws the result away and then goes on to evaluate the right operand which gives the result of the entire expression. This is commonly used to sequence operations involving side effects, such as `puts "Input:" $$ gets`.

- The special form `catch` evaluates an expression (given as the second, call-by-name argument) and returns its value, unless an exception occurs in which case the first (call-by-value) argument, the *handler*, is applied to the reported exception value. Exceptions may be raised through the runtime system in case of abnormal error conditions such as failed conditionals and matches, or explicitly with the built-in `throw` function. Example:

  ```
  > catch handler (throw some_value);
  handler some_value
  ```

  Exception handlers may be nested, in which case control is passed to the innermost handler (which may also throw on exceptions it doesn't want to handle). Unhandled exceptions are reported by the interpreter.

  Note that exception handling is an imperative programming feature which falls outside the realm of "pure" term rewriting, but it can save a lot of trouble if you need to handle error conditions deep inside a function. Another typical use case are non-local value returns; see Section 8.8 for a practical example.

- The special form `quote` quotes an expression, i.e., `quote` *x* (which may also be written as `'`*x*) returns just *x* itself without evaluating it. This facility should be well familiar to Lisp programmers. The built-in function `eval` 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
  ```

  It is worth noting here that Pure differs from Lisp in that local variables are substituted even inside quoted expressions. This makes it possible to fill in the variable parts in a quoted "template" expression quite easily, without an arguably complex

tool like Lisp's "quasiquote" operation; the downside is that local variables cannot be quoted. The Pure manual discusses various techniques for working with Pure's `quote`, so please refer to the manual for more details.

- Another way to defer the evaluation of an expression is the special form *x*`&` which is called a *thunk* or a (lazy) *future*; see below for details.

Pure's variation of thunks was adopted from Alice ML [17] to support *lazy evaluation*, as opposed to *eager evaluation* which corresponds to the normal call-by-value evaluation order. Thunks are written as a postfix application *x*`&` which turns its argument *x* into a kind of parameterless closure to be evaluated when *x* is first needed. The value is then also *memoized* so that subsequent accesses just retrieve the already computed value. This combination of call-by-name and memoization is also known as *call-by-need* evaluation. E.g., the following expression "thunks" the computation `square (6*7)` until `x+1` "forces" its evaluation, after which `x` has been memoized and is now readily available:

```
> let x = square (6*7)& with square x = x*x end; x;
#<thunk 0x7f7c91e71188>
> x+1; x;
1765
1764
```

Thunks can be employed to implement all kinds of lazy data structures. One particularly important example are *lazy lists*, also known as *streams* in the functional programming literature. Basically, a stream is a list `x:xs&` whose tail has been thunked. This enables you to work with infinite lists (or finite lists which are so huge that you would never want to keep them in memory in their entirety). E.g., here's one way to define the infinite stream of all Fibonacci numbers:

```
> fibs = fibs 0L 1L with fibs a b = a : fibs b (a+b) & end;
> fibs;
0L:#<thunk 0x7f6be902d3d8>
```

The prelude has full support for lists with thunked tails so that most common list operations such as concatenation, indexing and list comprehensions work with streams in a lazy fashion. So, for instance, we may retrieve a finite segment of the Fibonacci stream using list slicing:

```
> fibs!!(0..14);
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L,55L,89L,144L,233L,377L]
```

Note that our Fibonacci stream is really infinite, although at any time only a finite segment of it is actually in memory. If you're patient enough, you can retrieve *any* member of this sequence:[6]

[6]One has to be careful, though, to prevent *memory leaks* which occur if streams are allowed to grow indefinitely due to memoization. This is also the reason why we have defined the stream as a parameterless function in this example, rather than binding it to a global variable. The stream is thus recomputed on the fly each time we need it, so that only a small part of it needs to be in memory at any time.

```
> fibs!1000000;
19532821287077577316320149475962563324 43... // lots of digits follow
```

The prelude also provides various operations for generating infinite stream values, including arithmetic sequences with infinite upper bounds. For instance, we can denote the list of all positive odd integers as follows (`inf` denotes infinity):

```
> let xs = 1:3..inf; xs;
1:#<thunk 0x7f89d58295c0>
> xs!!(0..10);
[1,3,5,7,9,11,13,15,17,19,21]
```

4 Definitions

Definitions at the toplevel of a Pure program take one of the following forms:

lhs `=` *rhs*`;` Rewriting rules always consist of a left-hand side pattern *lhs* (which must be a simple expression, cf. Section 3.5) and a right-hand side *rhs* (which can be any kind of Pure expression as described in the previous section). There are some variations of the form of rewriting rules which will be discussed in Section 4.2.

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

`type` *lhs* `=` *rhs*`;` Another special form of rewriting rule used in *type definitions*.

`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. Unlike variables, **`const`** symbols can only be defined once, and thus their values do not change during program execution.

4.1 The Global Scope

In contrast to local functions and variables introduced with **`with`** and **`when`**, the constructs listed above define symbols with *global scope*. To facilitate interactive usage, the global scope is *dynamic* in Pure. This differs from the lexical scoping discussed in Section 3.8 in that the scope of each global definition extends from the point where a function, macro, type, variable or constant is first defined, to the next point where the symbol is redefined in some way. Dynamic scoping 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
```

The dynamic global scope is mainly a convenience for interactive usage. But it 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. When a toplevel expression is evaluated, it will always use the definitions of global functions, variables, etc. in effect at this point in the program.

4.2 Rule Syntax

All global and local definitions in Pure share the same kind of basic rule syntax *lhs* = *rhs* with a pattern on the left-hand side and an arbitrary expression on the right-hand side.[7] However, the meaning of this construct depends on the context. There are two different kinds of rules being used in Pure:

- *Rewriting rules:* These are used to define functions, macros and types, and are executed "from left to right" when evaluating an expression, by "reducing" the left-hand side to the corresponding right-hand side. Lambdas and the rules used in **case** expressions work in a similar fashion, although the applied function is anonymous in this case and not mentioned in the patterns. If multiple rules are present, they are considered in the order in which they are written, and the first matching rule is picked. Rewriting rules can also be augmented with *guards*, and repeated left-hand or right-hand sides can be "factored out", as described below.

[7] Lambda notation slightly deviates from this, since it uses the syntax *lhs* -> *rhs* adopted from Haskell. But otherwise it works in the same fashion.

- *Simple rules:* These rules are also called *pattern bindings*. They are executed "from right to left"; the right-hand side is evaluated and then matched against the left-hand side pattern, in order to bind the variables in the pattern. This kind of rule is used in the **`let`**, **`const`** and **`when`** constructs, as well as in the generator clauses of comprehensions. Each rule denotes a separate definition here.

The following special constructs are only allowed in rewriting rules. (While type and macro definitions use the same general format, the rule syntax is more restricted there, see Sections 4.6 and 4.7 for details.)

- Guards: A *guarded equation* has the form *lhs* `=` *rhs* **`if`** *guard*`;` This indicates that the equation is only applicable if the guard evaluates to a non-zero integer. An exception is raised if the guard doesn't evaluate to a machine int. The guard may be followed by **`with`** and **`when`** clauses whose scope extends to *both* the right-hand side and the guard of the rule.

- Multiple right-hand sides: Repeated left-hand sides can be factored out, using the syntax *lhs* `=` rhs_1`;` `=` rhs_2`;` ... This expands to a collection of equations for the same left-hand side.

- Multiple left-hand sides: Repeated right-hand sides can be factored out, using the syntax lhs_1 `|` lhs_2 `|` ... `=` *rhs*`;` This expands to a collection of equations for the same right-hand side.

Here's another definition of the factorial, to illustrate a typical use of multiple right-hand sides and guards:[8]

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

Multiple left-hand sides occur less frequently, but they can be useful if you need different specializations of the same rule with different type tags on the left-hand side. For instance:

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

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

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

[8]The **`otherwise`** keyword in the second equation denotes an empty guard. This is just syntactic sugar, but it often improves readability since it points out the default case of a definition.

Type rules are a case where the "|" notation is used quite often. Note that type definitions are in fact just definitions of special predicate functions in disguise. In addition, the right-hand side can be omitted if it is just the constant `true`, in which case the members of the type are simply the instances of the given patterns. This makes it possible to write the definition of an *algebraic type* (which consists entirely of constructor patterns) in the following style:

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

In the **`let`**, **`const`** and **`when`** pattern binding constructs, the left-hand side can be omitted if it is the anonymous variable '_', indicating that you don't care about the value. The right-hand side is still evaluated, if only for its side-effects. This is used most often in conjunction with **`when`** clauses, e.g., to implement sequential prompt/input interactions like the following:

```
> using system;
> s when puts "Enter a value:"; s = gets end;
Enter a value:
99
"99"
```

Here is another typical example which prints intermediate results for debugging purposes:

```
> using math, system;
> solve p q = -p/2+sqrt d,-p/2-sqrt d if d>=0
> when d = p^2/4-q; printf "The discriminant is: %g\n" d; end;
> solve 4 2;
The discriminant is: 2
-0.585786437626905,-3.41421356237309
> solve 2 4;
The discriminant is: -3
solve 2 4
```

Note that this works because, as explained in Section 3.7, the individual definitions in a **`when`** clause are executed in sequence. This makes it possible to use **`when`** as a general-purpose sequencing construct similar to Lisp's special form `prog` (but without `prog`'s unstructured features such as "gotos" and non-local "returns").

4.3 Function Definitions

Functions are defined by a collection of equations, using the rewriting rule syntax described in the previous subsection. In this case, the left-hand side of the equation consists of the function symbol, possibly followed by some argument patterns. Here are some examples:

```
// A simple definition with just one equation.
square x   = x*x;

// The Ackerman function.
ack x y    = y+1 if x == 0;
           = ack (x-1) 1 if y == 0;
           = ack (x-1) (ack x (y-1)) otherwise;

// Sum the elements of a list.
sum []     = 0;
sum (x:xs) = x+sum xs;
```

When the interpreter evaluates a function application, the equations for the given function are tried in the order in which they are written. The first equation whose left-hand side matches (and whose guard evaluates to a nonzero value, if applicable) is used to rewrite the expression to the corresponding right-hand side, with the variables in the left-hand side bound to their corresponding values. This means that "special case" rules must be given before more general ones, as shown in the `ack` example above.

The `sum` example shows how to define a function by pattern-matching, in order to "deconstruct" a structured argument value. This also works with user-defined data structures, for which the programmer may introduce new constructor symbols in an ad-hoc fashion (cf. Section 3.2). For instance, here's how to implement an insertion operation which can be used to construct a binary tree data structure made from the `bin` and `nil` constructor symbols:

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

Note that `nil` needs to be declared as a nonfix symbol here, so that the compiler doesn't mistake it for a variable. 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 generic accumulation function `foldl` is defined in the prelude as follows:

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

Since operators are just function symbols in disguise, they can be used on the left-hand side of equations as well. For instance, here is how you can define a lexicographic comparison on lists:

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

Pure doesn't enforce that you specify all equations of a global function in one go; they may actually be scattered out through your program, and even over different source files. The compiler only checks that all equations for a given function agree on the number of function arguments. 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. Pure supports both *parametric* and *ad-hoc polymorphism*, and you can also mix both styles. An example of parametric polymorphism is the following generic rule for the `square` function which applies to any argument `x` whatsoever:

```
> square x = x*x;
> square 99;
9801
> square 99.0;
9801.0
> square (a+b);
(a+b)*(a+b)
```

Instead, you can also write separate rules for different argument types, which gives you the opportunity to adjust the definition for each type of argument. For example, the prelude defines the '`*`' operator so that it works with different types of numbers. We might want to be able to also "multiply" a string by a number, like in Python, so let's add a definition for it:

```
> 5*6;
30
> 5*6.0;
30.0
> 5*"abc";
5*"abc"
> n::int * s::string = strcat [s | i=1..n];
> 5*"abc";
"abcabcabcabcabc"
```

This is an example of ad-hoc polymorphism, better known as *function overloading*. Note, however, that in contrast to overloaded functions in statically typed languages such as C++ and Haskell, there's really only *one* '`*`' function here; the dispatching needed to pick the right rule for the given argument is done by pattern matching at runtime.

Pure gives you a lot of leeway in writing your definitions. Most functional languages enforce the *constructor discipline*, which demands that only "pure" constructors

(i.e., function symbols without defining equations) should be used in the argument patterns of a function definition. Pure does *not* have this restriction; in fact it doesn't distinguish between "defined" functions and constructors at all. In particular, this allows you to have so-called *constructor equations*. For instance, suppose that we want lists to automatically stay sorted. In Pure we can do this by simply adding the following equation for the list constructor ':'.

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

In the same vein, you can also deal with algebraic identities in a direct fashion. For instance, let's try 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;
> (a+b)*(a+b);
a*a+a*b+b*a+b*b
```

Note that none of this is possible in Haskell and ML with their segregation of defined functions and data constructors. You'll basically have to write your own little term rewriting interpreter in those languages if you want to do such calculations.

Pure also provides a way to encapsulate such sets of algebraic simplification rules in a **with** clause, so that their scope is confined to a particular expression and different rule sets can be applied in different situations. This is done with a special predefined `reduce` macro which can be used in a way similar to Mathematica's `ReplaceAll` function. For instance:

```
> run // restart the interpreter to clear the above rules
> reduce ([13,7,9,7,1]+[1,9,7,5]) with x:y:xs = y:x:xs if x>y end;
[1,1,5,7,7,7,9,9,13]
> expand = reduce with (a+b)*c = a*c+b*c; a*(b+c) = a*b+a*c; end;
> factor = reduce with a*c+b*c = (a+b)*c; a*b+a*c = a*(b+c); end;
> expand ((a+b)*2);
a*2+b*2
> factor (a*2+b*2);
(a+b)*2
```

Before you run off and start programming your own computer algebra system now, be warned that term rewriting is just a small part of that. Out of the box, Pure doesn't offer any of the more advanced algorithms such as polynomial factorization and symbolic integration which make computer algebra really useful. Nevertheless, the kind of symbolic manipulations sketched out above can be pretty handy in "ordinary" code as well; see Section 8.10 for a practical example.

4.4 Variable Definitions

Variables may occur free on the right-hand sides of function definitions, in which case they can be given values with **let**:

```
> foo x = c*x; foo 21;
c*21
> let c = 2; foo 21;
42
```

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
```

Pattern matching works in variable definitions as usual:

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

4.5 Constant Definitions

The definition of a constant looks like a variable definition, using the **const** keyword in lieu of **let**:

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

In contrast to a global variable, a constant cannot change its value once it is defined, so its value can be substituted directly into subsequent definitions, as shown above. A constant can also be declared as **nonfix**, in which case its value also gets substituted into the left-hand side of equations. This is the case, in particular, for the predefined constants `true` and `false` which are declared **nonfix** in the prelude:

```
> show true false
nonfix false;
const false = 0;
nonfix true;
const true = 1;
> check x = case x of true = "yes"; false = "no"; _ = "dunno" end;
> show check
```

```
check x = case x of 1 = "yes"; 0 = "no"; _ = "dunno" end;
> map check [true,false,99];
["yes","no","dunno"]
```

Note that declaring a symbol **nonfix** makes it "precious", i.e., the symbol then cannot be used as a (local) variable any more. Therefore **const** symbols are almost never declared **nonfix** in the standard library; the predefined truth values are a notable exception.

4.6 Type Definitions

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

Syntactically, a type definition is a collection of rewriting rules which are each prefixed with the `type` keyword. No multiple right-hand sides are allowed in these definitions, but multiple left-hand sides are ok. Also, since a type definition actually defines a predicate denoting the terms belonging to the type, at most one argument is permitted on the left-hand side, and the result of invoking the predicate on a given term should be a truth value indicating whether the term belongs to the type or not. (If the result is anything but a nonzero machine integer, the predicate fails.)

In the simplest case, a type may just match a given left-hand side pattern. For instance:

```
type zero 0 = true;
```

This type consists of the constant (machine int) 0 and nothing else. As already mentioned, if the right-hand side is just `true` then it can also be omitted:

```
type zero 0;
```

This kind of notation is convenient for any kind of "algebraic" type which consists of a collection of constructor symbols with different arities. Note that the type symbol has to be repeated for each type rule or constructor pattern. For instance:

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

In general, the right-hand side of a type rule may be any expression returning a truth value. For instance, the following type denotes the positive machine ints.

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

In either case, the type symbol can then be used as a type tag on the left-hand side of other definitions, as described in Section 3.5. For instance, the `nat` type is used in the

following definition of the factorial in order to ensure that the argument is a positive integer (note that this definition would otherwise loop on zero or negative arguments).

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

New type rules can be added at any time. For instance, we can extend the `nat` type to bigints by just adding another type rule:

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

Without any further ado, our definition of the `fact` function now works with positive bigints, too:

```
> fact 30L;
265252859812191058636308480000000L
```

Type definitions can also be recursive. The compiler optimizes simple kinds of recursive type definitions so that the type check can be done in constant stack space, if possible. For instance, the type of proper lists is defined in the prelude as follows:

```
type rlist [] | rlist (x : xs::rlist);
```

Note that such a recursive type check needs linear time. Since type checks are done at runtime, it may thus become a serious performance hog; if used in a careless manner, it may easily turn a linear time algorithm into a quadratic one. (The same considerations apply to types defined by arbitrary predicates, unless they can be computed in constant time.) For instance, the following should be avoided:

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

One way to deal with such situations is to confine the type check to a so-called "wrapper" function which checks the type *once* for the entire list and then proceeds to call another "worker" function which implements the real algorithm without further type checks on the list argument:

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

Type *aliases* can be defined by omitting the left-hand side parameter and putting the target type symbol on the right-hand side. This is commonly used for numeric types, to document that they actually stand for special kinds of quantities:

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

The right-hand side can also be an existing ordinary predicate instead. In particular, this may also be a curried function or operator section which expects exactly one additional parameter. For instance, we might define the type of all positive numbers in a generic way as follows:

```
type positive = (>0);
```

Conversely, a type symbol can be converted to an ordinary predicate with the `typep` function:

```
> map (typep positive) [-1,0,1];
[0,0,1]
```

The right-hand side of a type definition may also be omitted altogether. This just declares the type symbol and makes it an empty type, so a proper type definition still has to be given later.

```
type thing;
```

Note that since a type definition may involve any unary predicate, any kind of relationship between two types is possible. One type may be a subtype of another, or they may be completely unrelated (i.e., disjoint), or some terms may belong to both types, while others don't. This gives the programmer a great amount of flexibility in data modelling.

Recent Pure versions also provide ways to define so-called *enumerated* and *interface types*. At present, these features are still a bit experimental and subject to change. So we only present a few code snippets below to whet your appetite; please refer to the Pure manual for details.

```
/* Enumerated data types are equipped with the usual operations, such as
   basic arithmetic, comparisons and arithmetic sequences. */

using enum;
enum day [sun,mon,tue,wed,thu,fri,sat];

/* Interface types are specified by their API, i.e., the operations they
   support. Here's an example of a stack data type */

interface stack with
  push s::stack x;
  pop s::stack;
  top s::stack;
end;

/* A possible implementation of the stack type in terms of lists. */

stack xs::list = xs;

push xs@[] x | push xs@(_:_) x = x:xs;
pop (x:xs) = xs; top (x:xs) = x;
pop [] | top [] = throw "empty stack";
```

4.7 Macro Definitions

Macros employ a restricted kind of rewriting rules (no guards, no multiple right-hand sides) which are applied by the interpreter at compile time. In Pure these are typically used to define custom special forms and to perform inlining of function calls and other kinds of source-level optimizations. 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).

For instance, the prelude defines the following macro 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 also 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));
```

You can also use macros to define your own special forms. The right-hand side of a macro rule may be an arbitrary Pure expression involving conditionals and block expressions. These special expressions are never evaluated during macro substitution, they just become part of the macro expansion. E.g., 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;
> count n = if n>0 then count(n-1) else n;
> timex (count 1000000);
0.4,0
```

This works because the call to `count` actually gets substituted into the **`when`** clause in the definition of `timex`:

```
> foo = timex (count 1000000);
> show foo
foo = (clock-t0)/1000000,y when t0 = clock; y = count 1000000 end;
```

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.

Please note that we barely scratched the surface here. In particular, Pure also lets you *quote* conditionals and block expressions in which case they become simple terms which can be manipulated by macros and ordinary functions in a direct fashion. Using some special library functions it is even possible to inspect and modify the rewriting rules of the running program, which gives the programmer access to powerful metaprogramming capabilities on a par with those provided by the Lisp programming language. These advanced features of Pure's macro system are beyond the scope of this guide, however, so we refer the reader to the Pure manual for details.

5 Programs and Modules

A Pure program is basically just a collection of definitions, symbol declarations and expressions to be evaluated. Pure doesn't support separate compilation right now, 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.

5.1 Modules

A Pure module is just an ordinary script file. A special kind of **`using`** declaration can be used to import one Pure script in another. In particular, this declaration allows you to import definitions from standard library modules other than the prelude. For instance:

```
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. 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.

5.2 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. To prevent name clashes, Pure allows you to put symbols into different user-defined namespaces. Like in C++, namespaces are completely decoupled from modules. Thus 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.

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. 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 5.3 below.) New symbols are always created as **`public`** symbols in the current namespace by default, so in this case we can also simply write:[9]

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

Also note that just the **`namespace`** keyword by itself in the last line switches back to the default namespace. For convenience, there's also a "scoped" namespace construct which indicates the extent of the namespace definition explicitly with a **`with ... end`** clause, so instead of the above you can also write:[10]

[9] This also works for any "defining" occurrence of a symbol on the left-hand side of an equation (such as the `foo` symbol in the example above), even if a symbol of the same name is already visible at the point of the definition. The Pure manual explains this in detail.

[10] Scoped namespaces can also be nested to an arbitrary depth, please check the Pure manual for details.

```
namespace foo with
foo x = x+1;
end;
namespace bar with
foo x = x-1;
end;
```

In any case, we can now refer to the symbols we just defined using qualified symbols of the form *namespace*`::`*symbol*:[11]

```
> 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 in Section 5.4.)

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

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
> using namespace;
```

A **`using namespace`** declaration without any namespace arguments gets you back to the default empty list of search namespaces. 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 (or up to the matching **`end`**, in the case of a scoped namespace declaration). 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.

[11] One of Pure's worst idiosyncrasies is that the `::` symbol is used for both type tags in patterns and namespace qualification. Thus a construct like `foo::int` may denote either a qualified identifier or a tagged variable (see Section 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. You'll also have to do this if either the variable or the type symbol is a qualified identifier.

Unless an absolute namespace prefix is used (see Section 5.4), symbols are always looked up first in the current namespace (if any), then in the search namespaces (if any), and finally in the default namespace. It is possible to list several namespaces in a **using namespace** declaration, and in order to prevent name clashes you can also specify exactly which symbols to import from a namespace, as follows:

```
using namespace name (sym1 sym2 ...);
```

For instance, consider:

```
namespace foo with
foo x = x+1;
end;
namespace bar with
foo x = x-1;
bar x = x+1;
end;
```

In this case, using both namespaces will give you a name clash on the `foo` symbol:

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

To resolve this, you might use a qualified identifier, but you can also selectively import just the `bar` symbol from the `bar` namespace:

```
> using namespace foo, bar (bar);
> foo 99;
100
> bar 99;
100
> bar::foo 99;
98
```

Recent Pure versions also provide a quick way to switch namespaces right in the middle of an expression using a so-called *namespace bracket*. This is a pair of outfix symbols which can optionally be associated with a namespace in its declaration; the outfix symbols must have been declared beforehand. For instance:

```
outfix « »;
namespace foo ( « » );
infixr (::^) ^;
x^y = 2*x+y;
namespace;
```

The code above introduces a `foo` namespace which defines a special variation of the `(^)` operator. It also associates the namespace with the « » brackets so that you can switch to the `foo` namespace in an expression as follows:

```
> «(a+b)^c»+10;
2*(a+b)+c+10
```

Note that the namespace brackets themselves are removed from the resulting expression; they are only used to temporarily switch the namespace to `foo` inside the bracketed subexpression. This works pretty much like a **`namespace`** declaration (so any active search namespaces remain in effect), but is limited in scope to the bracketed subexpression and only gives access to the public symbols of the namespace (like a **`using namespace`** declaration would do). The rules of visibility for the namespace bracket symbols themselves are the same as for any other symbols, so they need to be in scope if you want to denote them in unqualified form.

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;
> baz 99; // this creates a new symbol 'baz' in the default namespace
baz 99
> secret::baz 99;
<stdin>, line 27: symbol 'secret::baz' is private here
```

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

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;
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:[12]

```
> old::foo 99;
100
> ::old::foo 99;
198
```

6 C Interface

Accessing C functions is dead easy in Pure. 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. Example:

[12] 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.

```
> 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);
```

An external function can also be imported under an *alias*:

```
extern double sin(double) = mysin;
```

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
```

The range of supported C types encompasses `void`, `bool`, `char`, `short`, `int`, `long`, `float`, `double`, as well as arbitrary pointer types, i.e.: `void*`, `char*`, etc. Pure strings and matrices can be passed for `char*`, `int*` and `double*` pointers, respectively. The precise rules for marshalling Pure objects to corresponding C types are explained in the Pure manual. In practice these should cover most kinds of calls that need to be done when interfacing to C libraries.[13]

When resolving 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; these are searched for on a user-configurable path, please see the Pure manual for details. For instance:

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

In a similar fashion you can also load LLVM bitcode (`.bc`) files. In this case you don't even have to bother with the **extern** declarations, the interpreter extracts these from the bitcode files and generates them automatically for you.

```
using "bc:myutils";
```

[13] Two useful addons available as separate packages are the `pure-ffi` module which adds some functionality not covered in Pure's built-in C interface (such as calling back from C into Pure) and the `pure-gen` script which makes it easy to generate the needed **extern** declarations for large C libraries.

Moreover, C code as well as code written in other languages with LLVM-enabled compilers can be inlined directly in Pure scripts, using the `%< ... %>` construct. In particular, this works with C, C++ and Fortran, using clang and the gcc dragonegg plug-in.[14] For instance, here is a little snippet which calls some C code to compute the gcd (greatest common divisor) of two numbers:

```
%<
int mygcd(int x, int y)
{
  if (y == 0)
    return x;
  else
    return mygcd(y, x%y);
}
%>

map (mygcd 25) (30..35);
```

7 The Interpreter

This section assumes that you already have the Pure interpreter up and running on your system. Sources and installation instructions can be found at `https://agraef.github.io/pure-lang/`. Binary packages and ports for a number of systems including Linux (Arch and Ubuntu), macOS and Windows are also available, please check the corresponding links on the Pure website for details.

7.1 Running the Interpreter

Use `pure -h` to get help about the command line options. Just the `pure` command without any command line parameters invokes the interpreter in interactive mode, so that you can enter definitions and expressions to be evaluated at the '`>`' command prompt. Exit the interpreter by typing either the `quit` command or the end-of-file character (`Ctrl-D` on Unix systems) at the beginning of the command line.

Some other important ways to invoke the interpreter are summarized below.

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

`pure -b` *script* ... Runs the given scripts in batch mode.

[14] Unfortunately, dragonegg is not maintained any more, so the Fortran inlining capability isn't of much use these days; it's still possible to load Fortran code from shared libraries, however. On a happier note, Pure continues to offer special support for Grame's functional DSP programming language Faust, including the capability to inline Faust code, please check the Pure manual for details.

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

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

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

The latter form of invocation is useful, in particular, in "shebangs" which allow you to run a script directly from the shell, with additional command line parameters being passed to the script. To these ends, simply add a line like the following at the beginning of your main script and make the script file executable. (This only works in Unix shells.)

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

With the `-c` option, you can also compile a script to a native executable which can be run without the interpreter. This needs the basic LLVM toolchain (specifically, `opt` and `llc`). It is also possible to create native assembler (`.s`) and object (`.o`) files which can be linked into other programs and libraries, or LLVM assembler (`.ll`) and bitcode (`.bc`) files which can be processed with the LLVM toolchain. Please refer to the Pure manual for details.

When running interactively, `-g` enables the built-in symbolic debugger. See Section 7.4 below for details.

The Pure distribution comes with an Emacs mode which lets you run the Pure interpreter in an Emacs buffer. Normally, the mode will be installed along with the interpreter; please check the Pure installation instructions for details. Add the following lines to your `.emacs` startup file (additional customization options are described at the beginning of the `pure-mode.el` 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)
```

Having your script file loaded in Emacs, you can then use the keyboard command `Ctrl-C Ctrl-K` to run the script interactively in an Emacs buffer. Pure mode has many more features which let you edit and test Pure scripts with ease, so please check the online documentation for more information.

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.

The Pure interpreter has support for "tags" files in both emacs ("etags") and vi ("ctags") format, which can be used to quickly locate the global declarations and definitions of a Pure script in editors and other utilities like the `less` program which provide this feature. Tags files can be created with the `--etags` and `--ctags` options of the interpreter, please see the Pure manual for details. Emacs Pure mode also provides the "Make Tags" command to create a tags file in Emacs.

Current versions of the interpreter also offer the `--check` option which does a quick syntax check of a script without executing code or producing output files. This option is particularly useful with "flycheck," the Emacs on-the-fly syntax checker. Details can be found in the Emacs section of the installation manual.

7.2 Interactive Commands

When the interpreter is running in interactive mode, you can just type your definitions and expressions to be evaluated at the '>' command prompt. Basic arithmetic, logical, string, list and matrix operations are defined in the standard prelude which is normally loaded on startup, so that you can start using the interpreter as a sophisticated kind of desktop calculator right away. For instance:

```
> 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]
```

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

```
> last ans;
55
```

The interpreter also understands the following special commands for interactive usage. These have to be typed on a line by themselves, starting with the command keyword in column one.[15] A closer description of the commands is available in the Pure manual, which can be invoked in the interpreter with the `help` command.

`!` *command* Shell escape.

`break` [*symbol* ...] Set breakpoints. See Section 7.4 below.

`bt` Print backtraces. See Section 7.4 below.

`cd` *dir* Change the current working dir.

[15] Thus you can "escape" normal Pure code by indenting a line with one or more spaces; this is necessary if an expression starts with an identifier which looks like a command word. Recent versions of the Pure interpreter also provide an alternative command syntax in which interactive commands are escaped by prefixing them with a special character at the very beginning of the line; please check the corresponding section in the Pure manual for details.

`clear` [*option* ...] [*symbol* ...] Purge the definitions of the given symbols (variables, functions, etc.). Also, `clear ans` clears the most recent result (see the description of `ans` above).

`del` [*option* ...] [*symbol* ...] Delete breakpoints and tracepoints. See Section 7.4 below.

`dump` [`-n` *filename*] [*option* ...] [*symbol* ...] Dump a snapshot of the currently defined symbols to a text file. The file is in Pure syntax, so it can be loaded again with the `run` command, see below.

`help` [*target*] Display the Pure manual, or some other bit of documentation. This requires an html browser (`w3m` by default, you can set a different browser program with the `PURE_HELP` or the `BROWSER` environment variable). Try `help online-help` for more information on the online help facility.

`ls` [*args* ...] List files (shell `ls`(1) command).

`mem` Print current memory usage. This reports the number of expression cells currently in use by the program, along with the size of the freelist (the number of allocated but currently unused expression cells).

`pwd` Print the current working dir (shell `pwd`(1) command).

`quit` Exit the interpreter.

`run` [`-g`|*script*] Source the given script file and add its definitions to the current environment. If the script file is omitted, `run` restarts the interpreter with all original options and arguments, which is handy to quickly reload a script after some source files have been changed. In the latter case, you may also add the `-g` option to indicate that the interpreter should be invoked with debugging support.

`show` [*option* ...] [*symbol* ...] Show the definitions of symbols in various formats.

`stats` [`-m`] [`on`|`off`] Print some statistics after an expression evaluation. By default, this just prints the cpu time in seconds for each evaluation. With the `-m` option you also get information about the expression memory used in a computation.

`trace` [*option* ...] [*symbol* ...] Set tracepoints. See Section 7.4 below.

Commands that accept options generally also understand the `-h` (help) option which prints a brief summary of the command syntax and the available options.

The `clear`, `dump` and `show` commands accept a common set of options for specifying a subset of symbols and definitions on which to operate. Options may be combined, thus, e.g., `show -mft` is the same as `show -m -f -t`. Some options specify optional numeric parameters; these must follow immediately behind the option character if present. When invoked without arguments, `-t1` is the default, which restricts the command to "temporary" definitions entered interactively at the command prompt.

`-c`, `-f`, `-m`, `-v`, `-y` Selects constant, function, macro, variable and type symbols, respectively. If none of these are specified, then all categories of symbols are selected.

`-g` Indicates that the following symbols are actually shell glob patterns and that all matching symbols should be selected.

`-p`[*flag*] Select only private symbols if *flag* is nonzero (the default), otherwise (*flag* is zero) select only public symbols (cf. Section 4). If this option is omitted then both private and public symbols are selected.

`-t`[*level*] Select symbols and definitions at the given level of definitions and above. The executing program and all imported modules (including the prelude) are at level 0, while interactive definitions are at the "temporary" level 1 and above (see Section 7.3 below). If *level* is omitted, it defaults to the current definitions level.

The `show` command provides you with a quick means to inspect the definitions of functions, variables, constants and macros. For instance:

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

The same information can also be written to a file with the `dump` command, which gives you a quick means to save the results of an interactive session. The written file is a Pure script which is ready to be loaded by the interpreter again.[16] By default, `dump` writes definitions to the `.pure` file which is sourced automatically when the interpreter starts up in interactive mode. You can also specify a different filename with the `-n` option and later source the file with the `run` command.

The `show` command understands a number of additional options which let you select an output format and choose the kind of information to print:

`-a` Disassembles pattern matching automata for the left-hand sides of rules.

`-d` Disassembles LLVM IR, showing the generated LLVM assembler code of a function.

`-e` Annotate printed definitions with lexical environment information (de Bruijn indices, subterm paths).

`-l` Long format, prints definitions along with the summary symbol information. This implies `-s`.

`-s` Summary format, print just summary information about listed symbols.

[16] Unfortunately, this isn't perfect because some kinds of Pure objects don't have a textual representation from which they could be reconstructed. But in any case you can load the saved script in a text editor and use it as a starting point for creating your own script file.

The `-a`, `-d` and `-e` options are most useful for debugging the interpreter itself, but the `-l` and `-s` options are helpful for ordinary usage. For instance:

```
> show -s
fib   fun 1 args, 1 rules
fibs  var
0 constants, 1 variables, 0 macros (0 rules), 1 functions (1 rules),
0 types (0 rules)
> show -l
fib   fun  fib n = if n<=1 then n else fib (n-2)+fib (n-1);
fibs  var  fibs = [0,1,1,2,3,5,8,13,21,34,55];
0 constants, 1 variables, 0 macros (0 rules), 1 functions (1 rules),
0 types (0 rules)
```

Note that some of the options (in particular, `-d`) may produce excessive amounts of information. By setting the `PURE_MORE` environment variable accordingly, you can specify a shell command to be used for paging, usually `more(1)` or `less(1)`. `PURE_LESS` does the same for evaluation results printed by the interpreter.

7.3 Definition Levels

There are a number of other commands which let you manipulate subsets of definitions interactively. To these ends, interactive definitions are organized as a stack of *levels*. The prelude and other loaded scripts are all at level 0, while interactive input starts in level 1. Each `save` command adds a new level, while each `clear` command (without any parameters) purges the definitions on the current level and returns you to the most recent level. For instance:

```
> save
save: now at temporary definitions level #2
> foo (x:xs) = x+foo xs;
> foo [] = 0;
> show
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
> clear
This will clear all temporary definitions at level #2.
Continue (y/n)? y
clear: now at temporary definitions level #1
> show
> foo (1..10);
foo [1,2,3,4,5,6,7,8,9,10]
```

It's also possible to have definitions in the current level override existing definitions on previous levels; the `override` command enables this option, `underride` disables it again. Also, the `run` command sources a script at the current level, so that you can quickly get rid of the loaded definitions again if you invoke `save` beforehand.

7.4 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
```

If you already have the interpreter running, you can also just type `run -g` at the prompt to restart it in debugging mode. In any case, 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 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).

At the debugger prompt ':', you can just keep on hitting the carriage return key to walk through the evaluation step by step, as shown above. The debugger also provides various other 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. Type the `h` command at the debugger prompt to get a list of the supported commands.

A third use of the debugger is to trace function calls. For that the interpreter provides the `trace` command which works similarly to `break`, but sets so-called "tracepoints" which only print rule invocations and reductions instead of actually interrupting the evaluation. For instance, assuming the same example as above, let's first remove the breakpoint on `fact` (using the `del` command) and then set it as a tracepoint instead:

```
> del fact
> trace fact
> fact 1;
** [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
** [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
++ [2] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 0
     --> 1
** [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
```

```
++ [2] (*): x::int*y::int = x*y;
     x = 1; y = 1
     --> 1
++ [1] fact: fact n::int = if n>0 then n*fact (n-1) else 1;
     n = 1
     --> 1
1
```

The `trace` command has a few options which allow you to control the amount of information printed; please see the Pure manual for details. The `break` and `trace` commands can also be used in concert if you want to debug some functions while only tracing others. Note that these are interpreter commands; to enter them at the debugger prompt, you'll have to escape them with the debugger's '`!`' command.

The debugger can also be triggered programmatically. To these ends, just place a call to the built-in `__break__` or `__trace__` function near the point in your code where you'd like to start debugging or tracing. This gives you much finer control over the precise location and the conditions under which the debugger should be invoked.

7.5 User-Defined Commands

The interpreter lets you define your own commands for interactive usage. Interpreter commands are implemented as string functions exported by the special `__cmd__` namespace. For instance, here's how you can define a command which just echoes its arguments:

```
> namespace __cmd__;
> echo s = s;
> echo Hello, world!
Hello, world!
```

Note that the command function receives the rest of the command line as a string. If it returns a string result, that string is printed by the interpreter. The command function may also throw an exception containing a string value, in which case an error message is printed instead.

You can put your command definitions into one of the interpreter's startup files (see below) so that they are always loaded when the interpreter is run in interactive mode. See the Pure manual for some useful examples.

7.6 Interactive Startup

When running in interactive mode, the interpreter automatically sources the following script files if they exist, in the given order: `~/.purerc`, `./.purerc`, `./.pure`. The `.pure` file is written by the `dump` command and thus is normally used to save and restore definitions in an interactive session. The other two files can be used by the programmer to provide any additional definitions for interactive usage.

8 Examples

Here are a few code snippets with brief descriptions so that you get an idea how Pure programs look like. More detailed explanations of these examples can be found in the Pure manual.

8.1 Hello, World

This is an utterly boring example, but it's customarily used to explain the necessary incantations to get programs to run in different language environments. So, without any further ado:

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

Of course you might just enter these lines at the prompt of the interpreter. But for the fun of it, let's put them into a script file `hello.pure`, say. You can run that script with the interpreter as follows:

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

Other options of the interpreter program are explained in Section 7. In particular, you can compile the program to a native executable as follows:

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

Note that Pure's batch compiler has some unusual properties. In particular, it actually *runs* the script while compiling it. This enables some powerful programming techniques such as partially evaluating the program at compile time; please see the Pure manual for details.

8.2 Fibonacci Numbers

The naive definition:

```
fib n = if n<=1 then n else fib (n-2) + fib (n-1);
```

This works only for small values of `n`, but there's a much better definition which uses the *accumulating parameters* technique. This cuts down the running time from exponential to linear, and makes the function tail-recursive so that it can be executed in constant stack space:

```
fib n = loop n 0 1 with
  loop n a b = loop (n-1) b (a+b) if n>0;
             = a otherwise;
end;
```

Example:

```
> map fib (0..20);
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

Note that if you want to compute some really huge Fibonacci numbers, you'll have to do the computation with bigints in order to prevent wrap-around:

```
fib n = loop n 0L 1L with
  loop n a b = loop (n-1) b (a+b) if n>0;
             = a otherwise;
end;
```

Example (the result has 208988 digits):

```
> fib 1000000;
19532821287077577316320149475962563324 43... // lots of digits follow
```

There's also a variation of the same algorithm which computes the stream (lazy list) of *all* Fibonnaci numbers (cf. Section 3.10):

```
> fibs = fibs 0L 1L with fibs a b = a : fibs b (a+b) & end;
> fibs; fibs!!(0..14);
0L:#<thunk 0x7f6be902d3d8>
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L,55L,89L,144L,233L,377L]
> fibs!1000000;
19532821287077577316320149475962563324 43... // lots of digits follow
```

8.3 Numeric Root Finder

Here is a basic implementation of the Newton-Raphson algorithm in Pure. Note that the `solve` function is to be invoked with the target function as the first and the initial guess as the (implicit) second argument.

```
let dx = 1e-8;  // delta value for the approximation of the derivative
let dy = 1e-12; // delta value for testing convergence
let nmax = 20;  // maximum number of iterations
solve f = loop nmax (improve f) with
  loop n f x = x if n <= 0;
    = if abs (x-y) < dy then y else loop (n-1) f y when y = f x end;
  improve f x = x - f x / derive f x;
  derive f x = (f (x+dx) - f x) / dx;
end;
```

Examples:

```
> sqrt x = solve (\t -> t*t-x) x;
> sqrt 2; sqrt 5;
1.4142135623731
2.23606797749979
> cubrt x = solve (\t -> t^3-x) x;
> cubrt 8;
2.0
```

8.4 Gaussian Elimination

This is a numeric algorithm to bring a matrix into "row echelon" form, which can be used to solve a system of linear equations:

```
gauss_elimination x::matrix = p,x
when n,m = dim x; p,_,x = foldl step (0..n-1,0,x) (0..m-1) end;

// One pivoting and elimination step in column j of the matrix:
step (p,i,x) j
= if max_x==0 then p,i,x
  else
    // updated row permutation and index:
    transp i max_i p, i+1,
    {// the top rows of the matrix remain unchanged:
     x!!(0..i-1,0..m-1);
     // the pivot row, divided by the pivot element:
     {x!(i,l)/x!(i,j)                 | l=0..m-1};
     // subtract suitable multiples of the pivot row:
     {x!(k,l)-x!(k,j)*x!(i,l)/x!(i,j) | k=i+1..n-1; l=0..m-1}}
when
  n,m = dim x; max_i, max_x = pivot i (col x j);
  x = if max_x>0 then swap x i max_i else x;
end with
  pivot i x       = foldl max (0,0) [j,abs (x!j)|j=i..#x-1];
  max (i,x) (j,y) = if x<y then j,y else i,x;
end;

// Helper functions:
swap x i j = x!!(transp i j (0..n-1),0..m-1) when n,m = dim x end;
transp i j p = [p!tr k | k=0..#p-1]
with tr k = if k==i then j else if k==j then i else k end;
```

It's also convenient to define an Octave-like print representation of matrices here:

```
using system;
__show__ x::matrix
= strcat [printd j (x!(i,j))|i=0..n-1; j=0..m-1] + "\n"
with printd 0 = sprintf "\n%10.5f"; printd _ = sprintf "%10.5f" end
when n,m = dim x end if dmatrixp x;
```

Example:

```
> let x = dmatrix {2,1,-1,8; -3,-1,2,-11; -2,1,2,-3};
> x; gauss_elimination x;
   2.00000   1.00000  -1.00000   8.00000
  -3.00000  -1.00000   2.00000 -11.00000
  -2.00000   1.00000   2.00000  -3.00000
[1,2,0],
   1.00000   0.33333  -0.66667   3.66667
   0.00000   1.00000   0.40000   2.60000
   0.00000   0.00000   1.00000  -1.00000
```

8.5 Rot13

While Pure encodes strings in a C-compatible way internally, most list operations in the Pure prelude carry over to strings, so that they can be used pretty much as if they were lists of (UTF-8) characters. Character arithmetic works as well. For instance, here's the rot13 encoding in Pure:

```
rot13 x::string = string (map rot13 x) with
  rot13 c = c+13 if "a" <= lower c && lower c <= "m";
          = c-13 if "n" <= lower c && lower c <= "z";
          = c otherwise;
  lower c = "a"+(c-"A") if "A"<=c && c<="Z";
          = c otherwise;
end;
```

Example:

```
> rot13 "The quick brown fox";
"Gur dhvpx oebja sbk"
> rot13 ans;
"The quick brown fox"
```

8.6 The Same-Fringe Problem

This is one of the classical problems in functional programming which has a straightforward recursive solution, but needs some thought if we want to solve it in an efficient

way. Consider a (rooted, directed) tree consisting of branches and leaves. To keep things simple, we may represent these structures as nested lists, e.g.:

```
let t1 = [[a,b],c,[[d]],e,[f,[[g,h]]]];
let t2 = [a,b,c,[[d],[],e],[f,[g,[h]]]];
let t3 = [[a,b],d,[[c]],e,[f,[[g,h]]]];
```

The *fringe* of such a tree is the list of its leaves in left-to-right order:

```
fringe t = if listp t then catmap fringe t else [t];
```

For instance:

```
> fringe t1; fringe t2; fringe t3;
[a,b,c,d,e,f,g,h]
[a,b,c,d,e,f,g,h]
[a,b,d,c,e,f,g,h]
```

The *same-fringe problem* is to decide, for two given trees, whether they have the same fringe. This can be solved without actually constructing the fringes using a generalization of the accumulating parameters technique called *continuation passing*. The following algorithm is a slightly modified transliteration of a Lisp program given in [2]. The continuations can be found, in particular, in the `g` parameter of `genfringe`; they provide a kind of callback function which gets invoked to process the rest of the tree after finishing the current subtree. The entire algorithm is tail-recursive, so it runs in constant stack space.

```
samefringe t1 t2 =
samefringe (\c -> genfringe t1 c done) (\c -> genfringe t2 c done) with
  done c = c [] done;
  samefringe g1 g2 =
    g1 (\x1 g1 -> g2 (\x2 g2 -> x1===x2 && (x1===[] || samefringe g1 g2)));
  genfringe [] c g = g c;
  genfringe (x:t) c g = genfringe x c (\c -> genfringe t c g);
  genfringe x c g = c x g;
end;
```

Example:

```
> samefringe t1 t2, samefringe t2 t3;
1,0
```

Henry Baker, who invented this technique, said himself that this style of programming is not "particularly perspicuous". It may be useful at times, but it's often much easier to solve these kinds of problems in an efficient way using lazy evaluation. For instance, here's a *much* simpler solution using streams (lazy lists):

```
> lazyfringe t = if listp t then catmap lazyfringe (stream t) else [t];
> lazyfringe t1 === lazyfringe t2, lazyfringe t2 === lazyfringe t3;
1,0
```

Note that our `lazyfringe` function differs from `fringe` only in that it converts the input tree `t` into a stream before handing it over to `catmap`. A simple syntactic equality check then suffices to decide whether the two trees have the same fringes. Using lazy evaluation makes sure that the fringes are only constructed as far as needed, giving a similar time and space efficiency as Baker's (admittedly much more ingenious) solution.

8.7 Prime Sieve

This is a version of Erathosthenes' prime sieve using list comprehensions. Please note that this algorithm is rather slow and thus unsuitable for generating large prime numbers. It also isn't tail-recursive and will thus run out of stack space if the primes get large enough. There are much better ways to implement this sieve, but they're also more complicated. This algorithm works ok for smaller primes, though, and is easy to understand:

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

Example:

```
> primes 100;
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

Using streams (cf. Section 3.10), we can also compute *all* primes as follows:

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

Example (hit `Ctrl-C` when you get bored):

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

8.8 8 Queens

This is an *n*-queens algorithm which uses a list comprehension to organize the backtracking search.

```
queens n       = search n 1 [] with
  search n i p = [reverse p] if i>n;
               = cat [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = ~any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

The board positions of the queens are encoded as lists of row-column pairs; a solution places all `n` queens on the board so that no two queens hold each other in check. (Note that for efficiency, the position lists are actually constructed in right-to-left order here, hence the call to `reverse` in the first equation for `search` which brings them into the desired left-to-right order.) E.g., let's compute the solutions for an 8×8 board:

```
> #queens 8; // number of solutions
92
> using system;
> do (puts.str) (queens 8);
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
[(1,1),(2,6),(3,8),(4,3),(5,7),(6,4),(7,2),(8,5)]
...
```

Here's a variation of the same algorithm which only returns the first solution:

```
queens n       = catch reverse (search n 1 []) with
  search n i p = throw p if i>n;
               = void [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = ~any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

This illustrates the use of `catch` and `throw` (cf. Section 3.10) to implement non-local value returns. As soon as the recursive `search` routine finds a solution, it gets thrown as an exception which is caught in the main `queens` routine. Another sublety worth noting is the use of `void` in the second equation of `search`, which effectively turns the list comprehension into a simple loop which suppresses the normal list result and just returns `()` instead. Example:

```
> queens 8;
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

8.9 AVL Trees

AVL trees are balanced search trees useful for sorting and searching. This example isn't in the manual, but it's included in the Pure distribution; the implementation follows [3].

```
nonfix nil;
type avltree nil | avltree (bin _ _ _ _);
avltreep = typep avltree;

avltree xs              = foldl insert nil xs;

null nil                = 1;
null (bin _ _ _ _)      = 0;

#nil                    = 0;
#(bin h x t1 t2)        = #t1+#t2+1;

members nil             = [];
members (bin h x t1 t2) = members t1 + (x:members t2);

member nil y            = 0;
member (bin h x t1 t2) y
                        = member t1 y if x>y;
                        = member t2 y if x<y;
                        = 1;

insert nil y            = bin 1 y nil nil;
insert (bin h x t1 t2) y
                        = rebal (mknode x (insert t1 y) t2) if x>y;
                        = rebal (mknode x t1 (insert t2 y));

delete nil y            = nil;
delete (bin h x t1 t2) y
                        = rebal (mknode x (delete t1 y) t2) if x>y;
                        = rebal (mknode x t1 (delete t2 y)) if x<y;
                        = join t1 t2;

/* Implement the usual set operations on AVL trees. */

t1 + t2                 = foldl insert t1 (members t2) if avltreep t1;
t1 - t2                 = foldl delete t1 (members t2) if avltreep t1;
t1 * t2                 = t1-(t1-t2) if avltreep t1;

t1 <= t2                = all (member t2) (members t1) if avltreep t1;
t1 >= t2                = all (member t1) (members t2) if avltreep t1;

t1 < t2                 = t1<=t2 && ~t2<=t1 if avltreep t1;
t1 > t2                 = t1>=t2 && ~t2>=t1 if avltreep t1;
```

```
t1 == t2                = t1<=t2 && t2<=t1 if avltreep t1;
t1 ~= t2                = ~t1==t2 if avltreep t1;

/* Helper functions. */

join nil t2             = t2;
join t1@(bin _ _ _ _) t2
                        = rebal (mknode (last t1) (init t1) t2);

init (bin h x t1 nil)   = t1;
init (bin h x t1 t2)    = rebal (mknode x t1 (init t2));

last (bin h x t1 nil)   = x;
last (bin h x t1 t2)    = last t2;

/* mknode constructs an AVL tree node, computing the height value. */

mknode x t1 t2          = bin (max (height t1) (height t2) + 1) x t1 t2;

/* height and slope compute the height and slope (difference between heights
   of the left and the right subtree), respectively. */

height nil              = 0;
height (bin h x t1 t2)  = h;

slope nil               = 0;
slope (bin h x t1 t2)   = height t1 - height t2;

/* rebal rebalances after single insertions and deletions. */

rebal t                 = shl t if slope t == -2;
                        = shr t if slope t == 2;
                        = t;

/* Rotation operations. */

rol (bin h x1 t1 (bin h2 x2 t2 t3))
                        = mknode x2 (mknode x1 t1 t2) t3;

ror (bin h1 x1 (bin h2 x2 t1 t2) t3)
                        = mknode x2 t1 (mknode x1 t2 t3);
```

```
shl (bin h x t1 t2)     = rol (mknode x t1 (ror t2)) if slope t2 == 1;
                        = rol (bin h x t1 t2);

shr (bin h x t1 t2)     = ror (mknode x t1 (ror t2)) if slope t2 == -1;
                        = ror (bin h x t1 t2);
```

Example:

```
> let t1 = avltree [17,5,26,5]; let t2 = avltree [8,17];
> members (t1+t2); members (t1-t2); t1-t2;
[5,5,8,17,17,26]
[5,5,26]
bin 2 5 (bin 1 5 nil nil) (bin 1 26 nil nil)
```

8.10 Unit Conversions

Converting units is another classical problem in scientific and engineering applications. Pure's symbolic evaluation capabilities discussed in Section 4.3 let us solve this problem in an interesting way. Note that we also employ the Newton-Raphson solver from above for converting between standard (SI) and arbitrary units. This makes the code rather generic, so that other kinds of units can be added quite easily.

```
// sample unit symbols
nonfix
  miles yards feet inches kilometers meters centimeters millimeters // length
  acres // area
  gallons liters // volume
  kilograms grams pounds ounces // mass
  seconds minutes hours // time
  fahrenheit celsius kelvin; // temperature

// base units
type unit miles | unit yards | unit feet | unit inches |
  unit kilometers | unit meters | unit centimeters | unit millimeters |
  unit acres | unit gallons | unit liters |
  unit kilograms | unit grams | unit pounds | unit ounces |
  unit seconds | unit minutes | unit hours |
  unit fahrenheit | unit celsius | unit kelvin;
// powers of base units
type unit (u::unit^n::int);
// complement type
type nonunit x = ~typep unit x;

// Determine the base and power of a unit.
```

```
base_of u::unit = case u of u^n = base_of u; _ = u end;
power_of u::unit = case u of u^n = n*power_of u; _ = 1 end;

// Split a dimensioned value in the normal form x*u1*...*un (see below) into
// its value (x) and unit (u1*...*un) parts.
value_of x = case x of x*u::unit = value_of x; _ = x end;
unit_of x = case x of
  x*u::unit = case unit_of x of 1 = u; v = v*u end;
  _ = 1;
end;

// Conversions to standard (SI) units.

standard_units = reduce with
  miles = 1760*yards;
  yards = 3*feet;
  feet = 12*inches;
  inches = 2.54*centimeters;
  kilometers = 1000*meters;
  centimeters = 0.01*meters;
  millimeters = 0.001*meters;

  acres = 43560*feet^2;

  gallons = 231*inches^3;
  liters = 1000*centimeters^3;

  grams = 0.001*kilograms;
  pounds = 453.59237*grams;
  ounces = pounds/16;

  minutes = 60*seconds;
  hours = 60*minutes;

  x*celsius = (x+273.15)*kelvin;
  x*fahrenheit = (5*(x-32)/9)*celsius;
end;

/* The following rules shuffle around units until a dimensioned value ends
   up in the normal form x*u1*...*un where each ui is a power of a base unit.
   It also sorts the units according to their names and reduces to powers of
   units where possible. Units in the denominator are expressed as negative
   powers; these always come last. */
```

```
reduce_units = reduce with
  x/u::unit = x*u^(-1);

  u::unit^0 = 1;
  u::unit^1 = u;
  (u::unit^n::int)^m::int = u^(n*m);

  x*u::unit*v::unit = x*u^(power_of u+power_of v) if base_of u === base_of v;

  x*(y*u::unit) = x*y*u;
  x*(y/u::unit) = x*y/u;
  x/(y*u::unit) = x/y/u;
  x/(y/u::unit) = x/y*u;

  u::unit*y::nonunit = y*u;
  u::unit/y::nonunit = 1/y*u;
  x*u::unit*y::nonunit = x*y*u;
  x*u::unit/y::nonunit = x/y*u;
  x/u::unit*y::nonunit = x*y/u;
  x/u::unit/y::nonunit = x/y/u;

  x*u::unit*v::unit = x*v*u if sgn (power_of u) < sgn (power_of v) ||
    sgn (power_of u) == sgn (power_of v) && str (base_of u) > str (base_of v);

  (x*u::unit)^n::int = x^n*u^n;

  x*u::unit+y*u = (x+y)*u;
  x*u::unit-y*u = (x-y)*u;
  -x*u::unit = (-x)*u;
end;

/* Normalize a dimensioned value, converting it to standard units. Note that
   you can just use reduce_units instead if you want to normalize the value
   without converting it. */

si x = reduce_units (standard_units x);

/* Convert a dimensioned value to any (possibly non-standard) units, reduced
   to normal form. Source and target units must be compatible. */

infix 1450 as;
```

```
x as u = reduce_units (y*u) when
  // Note that we invoke the solver with the precomputed normal form v of the
  // target unit instead of u itself. The results shouldn't differ but this
  // will presumably speed up the computation.
  v = unit_of (reduce_units (1*u));
  y = solve v (value_of x);
end if unit_of y === unit_of x when
  // Normalize x so that it uses standard units.
  x = reduce_units (standard_units x);
  // Also normalize the target unit so that we can check that source and
  // target are compatible.
  y = reduce_units (standard_units (1*u));
end with
  solve u x = solve f x with
    // The target function: To solve for a given unit u, compute its SI value
    // and subtract the target value x.
    f y = value_of (si (y*u)) - x;
    // Newton-Raphson root finder. You might have to adjust the dx, dy and
    // nmax values below to make this work.
    solve f = loop nmax (improve f) with
      loop n f x = x if n <= 0;
        = if abs (x-y) < dy then y else loop (n-1) f y when y = f x end;
      improve f x = x - f x / derive f x;
      derive f x = (f (x+dx) - f x) / dx;
    end when
      dx = 1e-8;  // delta value for the approximation of the derivative
      dy = 1e-12; // delta value for testing convergence
      nmax = 50;  // maximum number of iterations
    end;
  end;
end;
```

Examples:

```
> si (1*feet^3/minutes+1*gallons/seconds);
0.0042573592272*meters^3*seconds^(-1)
> ans as liters/minutes;
255.441553632*liters*minutes^(-1)
> si ans;
0.0042573592272*meters^3*seconds^(-1)
> ans as inches^3/hours;
935280.0*inches^3*hours^(-1)
```

```
> si (1*yards^2);
0.83612736*meters^2
> ans as inches^2;
1296.0*inches^2

> 30*celsius as fahrenheit;
86.0*fahrenheit
```

I think that this last example illustrates the advantages of a language based on term rewriting really well. The symbolic manipulations needed to normalize unit values are at the heart of the program, and there's no need to jump through hoops to get that functionality since it's built right into the language. These symbolic evaluation capabilities, paired with dynamic typing, make for an incredibly flexible and powerful computing tool. Pure clearly has an edge over statically typed functional programming languages like Haskell and ML there. On the other hand, these excel in their own areas, notably better type safety and more efficient code through automatic type inference.

9 References

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

[2] H. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, 1993.

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

[4] R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *Proc. LISP Conference*, pages 136–143, Stanford University, 1980.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] A. Gräf. *The Q Programming Language*. `http://q-lang.sf.net`, 2010.

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

[12] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer, 1985.

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

[14] C. Lattner et al. The LLVM compiler infrastructure. `http://llvm.org`, 2012.

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

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

[17] 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`.

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

[19] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Proc. International Colloquium on Automata Languages and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*, pages 247–260, 1992.

[20] 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. Addison-Wesley, NY, 1990. See `http://miranda.org.uk`.

[21] W. van Oortmerssen. *Concurrent Tree Space Transformation in the Aardappel Programming Language*. PhD thesis, University of Southampton, UK, 2000. See `http://strlen.com/aardappel-language`.

[22] Wikipedia article "Functional programming". `http://en.wikipedia.org/wiki/Functional_programming`, 2012.

10 Index

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 Section 3.

script : {*item*}

item :
- **namespace** [*name*] [*brackets*] ;
- **namespace** *name* [*brackets*] **with** *item* {*item*} **end**;
- **using namespace** [*namespec* {, *namespec*}] ;
- **using** *name* {, *name*};
- **interface** *qualified-identifier* **with** {*interface-item*} **end**;
- [*scope*] **extern** *prototype* {, *prototype*};
- *declarator qualified-symbol* {*qualified-symbol*};
- **let** *simple-rule*;
- **const** *simple-rule*;
- **def** *macro-rule*;
- **type** *type-rule*;
- *rule*;
- *expr*;
- ;

interface-item :
- *pattern*
- **interface** *qualified-identifier*;

name : *qualified-identifier* | *string*

brackets : (*left-op right-op*)

namespec : *name* [({*symbol*})]

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

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

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

precedence : *integer* | `(` *op* `)`

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

parameters : *parameter* {`,` *parameter*} [`, ...`]
| `...`

parameter : *c-type* [*identifier*]

c-type : *identifier* {∗}

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

type-rule : *pattern* {| *pattern*} [= *expr* [*guard*]]

macro-rule : *pattern* {| *pattern*} = *expr*

simple-rule : *pattern* = *expr* | *expr*

pattern : *simple-expr*

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

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 op simple-expr*
| *op simple-expr*
| *simple-expr op*
| *application*

application : *application prim-expr*
| *prim-expr*

prim-expr : *qualified-identifier* [`::` *qualified-identifier* | `@` *prim-expr*]
| *qualified-symbol*
| *number*
| *string*
| `(` *op* `)`
| `(` *left-op right-op* `)`
| `(` *simple-expr op* `)`
| `(` *op simple-expr* `)`
| `(` *expr* `)`
| *left-op expr right-op*
| `[` *exprs* `]`
| `{` *exprs* {`;` *exprs*} [`;`] `}`
| `[` *expr* `|` *simple-rules* `]`
| `{` *expr* `|` *simple-rules* `}`

exprs : *expr* {`,` *expr*}

rules : *rule* {`;` *rule*} [`;`]

simple-rules : *simple-rule* {`;` *simple-rule*} [`;`]

op : *qualified-symbol*

left-op : *qualified-symbol*

right-op : *qualified-symbol*

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*} [*exponent*]

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

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

symbol : *identifier* | *special*

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⟩

The Pure language has a number of reserved keywords which cannot be used as identifiers. These are:

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

Note that the character repertoire available for the lexical entities *letter*, *punct* and *char* depends on the basic character set that you use. The current implementation only supports the UTF-8 encoding, so you either have to use that (most text editors should support UTF-8 nowadays) or confine your scripts to 7 bit ASCII (which is a subset of UTF-8). In addition to the ASCII punctuation symbols, Pure considers the following extended Unicode characters 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 a few additional blocks of Unicode symbols[17], which should cover almost everything you'd ever want to use in operator symbols. All other extended Unicode characters are considered as letters.

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 reserved 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 also provides escape sequences for all Unicode characters, which lets you use the full Unicode set in strings even if your text editor only supports ASCII; please see Section 2 for details.

[17] Just for the record, these are: 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. Thanks are due to John Cowan who suggested this scheme which greatly simplifies Pure's lexical syntax.

B Term Rewriting

In this appendix we take a brief look at the basic notions of term rewriting theory which underly Pure's model of computation. This material shouldn't be necessary to work with Pure on a practical level, but it will be interesting for programming language designers and theorists and anyone else who would like to have an exact and formal (if somewhat abstract) operational model of how Pure works as a term rewriting engine.

B.1 Preliminaries

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_Σ is the set of all variable symbols in Σ.[18] The (free) *term algebra* over the signature Σ is the set of terms defined recursively as:

$$T_\Sigma = \{f\, t_1 \cdots t_n \mid f \in \Sigma_n, t_i \in T_\Sigma\}$$

A *term rewriting rule* is an ordered pair of terms $p, q \in T_\Sigma$, denoted $p \to q$. In order to describe the meaning of these, we also need the notion of a *substitution* σ 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 explicitly 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 σ (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 $\Box$ such that $t = s[\Box \to u]$. That is, t is just s with the subterm u at the position indicated by $\Box$.

B.2 Basic Term Rewriting

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 $t = s[\Box \to u]$, where $u = \sigma(p)$. Then we can rewrite t to $t' = s[\Box \to v]$ 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 subterm $u = f\, (g\, x)\, c$ of the subject term $t = g(f\, (g\, x)\, c)$, where the context is $s = g\, \Box$ and the matching substitution is $[x \to g\, x, y \to c]$, we obtain $t' = g(h\, (g\, x))$.

[18] Note that term rewriting theory usually employs uncurried function applications, but the curried notation used by Pure 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.

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

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 $t \rightarrow_R t'$. If $t \rightarrow_R^* t'$ such that t' is in normal form, then we also call t' a *normal form of* t.

B.3 Term Rewriting and Equational Logic

The basic term rewriting model sketched out above has applications in mathematical logic. To see how, we need to consider the notion of a Σ-*algebra* A where each $f \in \Sigma_n \setminus X_\Sigma$ is associated with an n-ary function $f^A : A^n \mapsto A$. Each *ground term* $t \in T_{\Sigma \setminus X_\Sigma}$ then corresponds to an element t^A of A which is defined recursively as follows:

$$(f\, t_1 \cdots t_n)^A = f^A(t_1^A, \ldots, t_n^A) \;\forall\; f \in \Sigma_n \setminus X_\Sigma,\; t_1, \ldots, t_n \in T_{\Sigma \setminus X_\Sigma}$$

An equation $u = v$ of ground terms u and v is said to *hold* in A iff $u^A = v^A$. More generally, if $u = v$ is an equation of arbitrary Σ-terms u and v (possibly containing variables), then $u = v$ is said to hold in A iff $\sigma(u)^A = \sigma(v)^A$ for each substitution σ such that both $\sigma(u)$ and $\sigma(v)$ are ground terms. Also, a *set* of equations E holds in A iff $u = v$ holds in A for each $u = v \in E$, which is written $A \models E$. We then also say that A is a *model* of E. If an equation $u = v$ holds in *every* model A of E, then we also say that $u = v$ is a *logical consequence* of E and write $E \models u = v$.

Now, given a term rewriting system R, we can look at the corresponding set of equations $E = \{p = q \mid p \rightarrow q \in R\}$. As it turns out, an equation $u = v$ holds in every model of E (i.e., $E \models u = v$) if and only if $u \leftrightarrow_R^* v$. This is also known as *Birkhoff's theorem*. Under certain circumstances, the rewriting system can then be used as a procedure for deciding whether two given terms are equal in all models by just comparing their normal forms. To make this work, the term rewriting system needs to be *terminating* (there are no infinite chains $u_0 \rightarrow_R u_1 \rightarrow_R u_2 \rightarrow_R \cdots$) and *confluent* ($\forall u, v_1, v_2 : u \rightarrow_R^* v_1, v_2 \Rightarrow \exists w : v_1, v_2 \rightarrow_R^* w$).

Unfortunately, these conditions are often not satisfied in practice. Normal forms need not always exist and even if they do, they might not be unique. In fact the termination and confluence properties are not even decidable, because term rewriting is a Turing-complete model of computation. Much of the deeper parts of term rewriting theory deals with precisely these issues. But if we want to retain Turing-completeness then we inevitably loose some of the simple and mathematically elegant equational semantics of term rewriting sketched out above.

B.4 Conditional and Priority Rewriting

Term rewriting systems can still be employed as a useful model of computation even if they're neither confluent nor terminating. To these ends, one usually extends the basic term rewriting calculus so that it becomes better suited as a programming language.

One extension that turns out to be practically useful are *conditional rewriting rules*, written $p \to q$ **if** c. The condition c, which is also called a *guard* in functional programming parlance, usually takes the form of a conjunction of formal equations in mathematical logic. But for our purposes we actually permit arbitrary terms $c \in T_\Sigma$ as guards. We also assume two special constant symbols $\text{true}, \text{false} \in \Sigma_0$ which denote the truth values. Now a conditional rule $p \to q$ **if** c can be applied to a redex $u = \sigma(p)$ in the same manner as before, but only if the condition is satisfied, i.e., $\sigma(c) \to_R^* \text{true}$. The reduction relation $\to_R$ is modified accordingly: we now have $t = s[\square \to u] \to_R t' = s[\square \to v]$ if $u = \sigma(p)$, $v = \sigma(q)$ and $\sigma(c) \to_R^* \text{true}$.[19]

Conditional rules allow you to define a function in a piecewise fashion, as one commonly does in mathematics. As an example, here is a conditional rewriting system for the factorial:

$$\begin{aligned} \text{fact}\, n &\to 1 \;\textbf{if}\; n \leq 0 \\ \text{fact}\, n &\to n \times \text{fact}\,(n-1) \;\textbf{if}\; n > 0 \end{aligned}$$

Another useful extension which goes well together with conditional rewriting is *term rewriting with priorities*. Here we equip R with a *priority order* $<$. A rewriting rule $r = p \to q$ or $r = p \to q$ **if** c may then only be applied to a given redex u if there's no other rule $r' < r$ in R which can be applied to the same redex. Priorities orders may be total (like the *textual order* where the rewriting rules are considered in the order in which they are written in the program) or partial (like the so-called *specificity order* where $r < r'$ if the left-hand side of r is an instance of the left-hand side of r'). Pure employs the textual order which is often considered more intuitive and is also used in mainstream functional languages such as ML and Haskell. (The specificity order also has its advantages, however, and has actually been used with great success in languages such as Aardappel [21] and Hope [4].)

Like conditional rules, rewriting with priorities makes it possible to write some definitions which aren't easily formulated in the basic term rewriting calculus, such as the following definition of the factorial:

$$\begin{aligned} \text{fact}\, 0 &\to 1 \\ \text{fact}\, n &\to n \times \text{fact}\,(n-1) \end{aligned}$$

Taken as an ordinary rewriting system, this system is non-terminating (it "loops" on the second rule), but it becomes usable as a program to compute the factorial of nonnegative numbers if we assume that the first rule takes priority over the second one.

[19] Pure actually requires that $\sigma(c) \to_R^* c' \in \{\text{true}, \text{false}\}$, otherwise the program is considered in error and an exception will be raised. Also note that Pure doesn't have a separate type for the truth values, so they are represented as machine integers, where 0 denotes false and any nonzero value denotes true.

Conditional rules are also frequently given priorities to allow for more succint definitions. The basic idea is that if the rules are always tried in the indicated order then each successive rule may assume that the negation of all previous guards holds. For instance, consider the following definition of the Ackerman function:

$$\begin{aligned}\text{ack } x\ y &\to 1 \textbf{ if } x \leq 0\\ \text{ack } x\ y &\to \text{ack } (x-1)\ 1 \textbf{ if } y \leq 0\\ \text{ack } x\ y &\to \text{ack } (x-1)\ (\text{ack } x\ (y-1))\end{aligned}$$

These rules only work as intended if they are tried exactly in the given order, because the second and third rules operate under the assumptions that $x > 0$ and $x, y > 0$, respectively; if the rules are considered in any other order then the resulting system doesn't terminate.

It goes without saying that, although the basic rewriting machinery still works the same, conditional and priority rewriting offer an amount of control over the rewriting process which makes this style notably different from the more declarative style of the basic term rewriting calculus with its purely equational semantics. But it is often convenient to write definitions this way and so most functional programming languages including Pure offer these features.

B.5 Reduction Strategy

Conditional and priority rewriting give the programmer better control over which rules can be applied on a given redex. But there is still a lot of non-determinism in the choice of redices in the rewriting process which makes these systems hard to use as a programming language. Therefore one usually imposes a suitable *reduction* or *evaluation strategy* which specifies the order in which redices are rewritten. When combined with a total rule priority order, this resolves all ambiguities in the rewriting process so that it becomes completely deterministic.

Common reduction strategies are the leftmost-innermost and leftmost-outermost strategies. *Leftmost-innermost* means that the leftmost redex which doesn't contain any other redex gets reduced first. It corresponds to "call-by-value" and hence is typically used in languages with eager evaluation. This strategy also lends itself to an efficient implementation where function applications are evaluated recursively by first evaluating the arguments of the function before the function is applied to its arguments.

The *leftmost-outermost* strategy always chooses the leftmost redex which isn't contained in any other redex. It corresponds to "call-by-name" and is used in languages based on lazy evaluation. This strategy is generally harder to implement in an efficient way, but is known to be optimal for rewriting systems based on the combinatorial calculus, in the sense that it never rewrites a redex unless it is really needed to determine the normal form of the target term.

B.6 Term Rewriting in Pure

Let us finally discuss how these notions apply to the Pure programming language. First, note that neither the symbol alphabet Σ nor the rewriting system R are static entities in Pure; they may evolve over time, as the programmer enters new rewriting rules interactively in the interpreter or calls metaprogramming functions in the Pure runtime system. Second, Pure also offers various convenient language constructs which aren't really part of term rewriting, so we have to describe how to map them to the basic calculus:

- Local function and variable definitions (Section 3.7): Local *variable* definitions are equivalent to applications of local functions, using the equivalences discussed in Section 3.7. Local *function* definitions can be reduced to global rewriting rules using the well-known technique of *lambda lifting* which eliminates all local functions and turns local environments into explicit function arguments [12].[20]

- Global variable and constant definitions (Sections 4.4 and 4.5): In the term rewriting model, these can be considered as rules of the form $c \to t$ with $c \in \Sigma_0 \setminus X_\Sigma$ which may be replaced with new rules when a new **`let`** binding becomes active.

- Type definitions (Section 4.6): Pure's types are term sets specified as unary predicates on terms, which in effect are global functions defined through ordinary term rewriting rules. Types may then be used as "tags" like `x::int` on the left-hand side of rewriting rules to restrict the types of subterms matched by the variables of a rule. In conditional rewriting, these type tags may be represented as additional conditions $c\ x$ on a rule, where c denotes the type predicate and x is the tagged variable.[21]

- Macro definitions (Section 4.7): These are just ordinary rewriting rules without conditions. What's special about the macro rules is that they are applied in a separate preprocessing stage at compile time, in order to rewrite plain terms and the right-hand sides of other (function and type) rewriting rules before any code is generated.[22]

The above comprises what may be called Pure's *purely functional core*, which can thus be described completely using the notions of conditional term rewriting with priorities.

[20] We should mention there that the Pure compiler uses an efficient lambda lifting algorithm which keeps the number of hidden arguments as small as possible, and also passes these extra parameters in an efficient way in order to reduce the runtime overhead needed to implement this feature.

[21] That's actually how Pure implements them in the general case. However, the built-in types like `int` and `double` are actually handled in a more efficient way by inlining the type checks in the pattern matching code.

[22] Since macro rules need to be applied to compile time expressions, the current implementation of the compiler includes a separate term rewriting interpreter for this purpose. This isn't as fast as native code, of course, but it's still reasonably efficient, given that the size of the involved terms and the amount of rewriting that needs to be performed in macro substitution is quite limited in practice.

B.7 The Evaluation Algorithm

Pure's basic evaluation strategy is leftmost-innermost (call-by-value). The terms which are potentially reducible are all of one of the following forms; all other kinds of terms, such as numbers, strings etc. are always irreducible in Pure.

- $t \in \Sigma_0$ is either a global variable or constant, or a parameterless function. This is the base case where t is evaluated in a direct fashion.
- $t = u\,v$ is a curried application, $u, v \in T_\Sigma$. In this case, u and v are evaluated recursively, in that order, yielding the normal forms $\bar{u}$ and $\bar{v}$, before the application $\bar{u}\,\bar{v}$ itself is evaluated.

It is easy to see that this recursive procedure evaluates leftmost-innermost redices first. There are exceptions from this evaluation order in Pure, however. If $t = u\,v$ is a *special form* (cf. Section 3.10), or a partial application of a special form, then v might be left unevaluated, i.e., it is treated like a normal form. The special form itself then takes care of evaluating the parameter as needed. (In Pure this is strictly a compile-time feature, i.e., the head symbol of t must be recognizable as a special form *at compile time* to make this happen. Otherwise, the standard call-by-value strategy is used.)

In either case we are now left with a term t whose subterms have already been evaluated recursively as needed, and we need to describe how t itself is to be evaluated. To these ends, we check whether t is reducible using any of the rewriting rules of the program. This is done using the following algorithm:

1. Match the subject term t against the left-hand sides p of rules $p \to q$ or $p \to q$ **if** c in R. If more than one rule matches, they are tried in the order in which they are listed in the program (i.e., using the textual rule order). If no rule matches then t is already in normal form and we're done.
2. Otherwise we obtain a matching substitution σ such that $\sigma(p) = t$. (This matching substitution is *minimal* in the sense that $\sigma(x) = x$ unless x actually occurs in p, which also implies that σ is determined uniquely by p and t.)
3. For conditional rules $p \to q$ **if** c, the guard $\sigma(c)$ is evaluated recursively using the matching substitution determined in step 2. If the result is false (zero), try the next rule (go back to step 1). If the result is true (a nonzero integer), proceed with step 4. Otherwise the program is in error and an exception is raised.
4. Recursively evaluate the right-hand side $\sigma(q)$ using the matching substitution determined in step 2.

These steps are actually interleaved with term construction so that intermediate results don't have to be constructed explicitly unless they are normal form terms. Moreover, step 1 also recursively evaluates "thunked" subterms (cf. Section 3.10) if the corresponding part of the subject term needs to be inspected during pattern matching.

Step 1 of this algorithm might seem inefficient, but luckily the 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 find all matching rules in one go [9]. In most cases the matching overhead is barely noticable, unless you discriminate over huge sets of heavily overlapping patterns. 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.

B.8 Primitives

This finally leaves us with the primitive (built-in and external) operations of the Pure language, which also includes the handful of built-in special forms discussed in Section 3.10. In the case of operations like arithmetic which work in a purely functional way, these could in principle be specified using appropriate rewriting systems. However, many primitives (including some built-ins of the Pure language, most notably exceptions) involve observable side effects and thus fall outside the rewriting model of computation. For our purposes it's most convenient to just consider all primitive functions as "oracles" which reduce to the corresponding function result in a single "spontaneous" rewriting step such as $3 + 4 \to 7$.

To obtain a complete formal semantics, one might specify the actual behaviour of the primitives by some other means such as denotational semantics. While this is theoretically possible, it is a monumental task and in any case beyond the scope of this manual, considering that Pure allows you to call any C function.

In practice, the Pure compiler inlines calls to some built-in operations, including arithmetic (if the argument types are known) and the built-in special forms, as native code. Other operations are implemented in the runtime or other 3rd party libraries and are called via Pure's C interface, which automatically handles the necessary conversions between Pure's term data structure and native data such as numbers and pointers. Most primitive operations are partial functions and thus their applications are treated like normal forms if arguments don't match, which gives the programmer the opportunity to specify his own rewriting rules to overload the primitive definitions (cf. Section 6).