A Concise Guide to Haskell

This is an updated version of my earlier A Concise Introduction to Gofer/Haskell.

Haskell is a *purely functional programming language;* this means:

- Functions are values; they can be given as arguments to functions, and returned as the value of functions.
- Functions have no side effects.
- Functions are *referentially transparent*: A function, given the same arguments, will always return the same value.
- No state information is kept; there are no assignment statements, and "variables" never change their value.

Haskell is quite different from most currently popular languages, and requires a different way of thinking about programs. However, since functional programming is uniquely suited to concurrent programming, and modern computers are less frequently single-processor machines, most languages have incorporated some aspects of functional programming.

Haskell is a particularly enjoyable language in which to program, as the language does so much to make it easy for the programmer. A lot of structural syntax simply isn't needed. Powerful operators allow for concise, yet still readable, programs. Although the language is strongly typed, the programmer hardly ever needs to specify types--Haskell figures out types for itself. Functions are first-class objects, and infinite data structures are supported. Haskell is more human-oriented and less computer-oriented than most programming languages and, as a result, is less efficient; as yet it has little practical application.

Haskell has an underlying uniform representation. But for convenience, it has been extended with syntactic sugar to provide familiar-looking representations for arithmetic, strings, lists, and the like.

Hoogle is excellent, online searchable documentation for many of Haskell's standard libraries.

Versions

The two most popular versions of Haskell are GHC, the Glasgow Haskell Compiler, at http://hackage.haskell.org/platform/, and Hugs, at http://www.haskell.org/hugs/. Both are available on all popular operating systems. GHC is more actively supported, and is the author's preferred implementation.

Basic operation

To run the GHC interpreter, type `ghci` at the command line. This starts a REPL

(Read-Eval-Print-Loop) at which you can enter expressions, and the interpreter will print the result of evaluating those expressions. Thus, if you type `2+2` at the `ghci` prompt, `ghci` will respond with `4`. The usual precedence rules apply, and parentheses may be used. For example, `2+3*4+5` means the same as `2+(3*4)+5`. Technically, the operands of a binary operator must be of the same type (for example, both integer or both floating point), but Haskell 98 is usually pretty good at assigning types, so you only occasionally have to make an explicit conversion.

There is no "main program"--from the interpreter, any function may be called.

The `ghci` interpreter will only process single-line expressions; multi-line expressions must be loaded from a file. To define a function directly in the interpreter, a "let expression" must be used, for example:

```
let sum x y = x + y
```

In addition to evaluating expressions, the Haskell interpreter recognizes certain *commands,* all of which begin with a colon. Here are some of the more important ones:

Command	Meaning
`:?`	Display a list of the available commands
`:l` *filename*	Load in definitions from the specified file
`:r`	Reload definitions from the previously specified file
`:m +` *module ... module*	Import the named modules
`:t` *expression*	Give the type of the expression as well as its value
`:q`	Quit

To call a function, write the name of the function followed by its arguments. No additional punctuation is used. For example,

```
sum 100 y
sqrt 2.0
```

Case is significant. Variable names and function names must begin with a lowercase letter and may contain letters, digits, underscores, and apostrophes. Type names must begin with a capital letter.

Binary infix operators (such as `+` and `>`) may be converted to prefix form by enclosing them in parentheses. Thus,

`(+) 2 2` is the same as `2 + 2`

Binary prefix operators (such as `mod`) can be written in infix form by enclosing them in backquotes. Thus,

`mod 100 3` is the same as `100 `mod` 3`

Comments

There are two kinds of comment in Haskell:

`--` Anything after a double hyphen, up to the end of the line.

`{-` Comments may be enclosed in braces with hyphens. `{-` These comments can be nested `-}` . `-}`

Types

Simple types

There are several simple types; here are some of the more common ones.

Primitive type	Representative values
`Int`	`-5, 0, 5`
`Integer`	`30414093201713378043612608166064768844377641568960512000000000000`
`Float`	`3.14159`
`Double`	`3.141592653589793`
`Char`	`'a', '\n', '\\', '\'', '\97', '\o77', '\xFF', '\DEL'`
`Bool`	`True, False`

Type descriptions

In the tables in the following sections, the notation `Char -> Int` means a function that takes a character argument and produces an integer result. A lowercase letter (usually `a`) is a type variable; it indicates *any* type. When it occurs more than once in a type description, for example `a -> a`, all occurrences indicate the *same* type. The notation `Num a => a -> a` indicates that `a` is restricted to be a numeric type.

The notation `Int -> Int -> Int` means a function that takes two integer arguments and produces an integer result. (The last type in the chain is always the result. The reason for this notation for types will be explained later.)

Types can be given aliases (names), for example, `type CardDeck = [Card]`. Careful use of aliasing can greatly improve the readability of type signatures.

In the REPL, the type of a function can be found by entering `:type` *`func`*. Infix operators must be enclosed in parentheses to make them into prefix functions, for example, `:type (+)`.

Functions on integers

Function	Type	Comment
`^`	`(Num a, Integral b) => a -> b -> a`	Raise to the power
`+ - *`	`Num a => a -> a -> a`	Infix add, subtract, multiply
`/`	`Fractional a => a -> a -> a`	Divide, always yields a real number, not an integer

`-`	`Num a => a -> a -> a`	Unary minus is an operator, not part of a number. Thus, `abs -3` parses as `(abs -) 3`, which is illegal; use `abs (-3)` instead.
`div quot mod`	`Integral a => a -> a -> a`	`div` rounds down (same as `/`); `quot` rounds toward zero; modulo
`gcd lcm`	`Integral a => a -> a -> a`	Greatest common divisor, least common multiple
`negate -`	`Num a => a -> a`	Negate, unary minus
`even odd`	`Integral a => a -> Bool`	Prefix tests for parity
`signum`	`Num a => a -> a`	-1, 0, or 1, according to whether the argument is negative, zero, or positive

Functions on floating point numbers

Function	Type	Comment
`+ - * / ^`	As above (Integer table)	Infix add, subtract, multiply, divide, exponentiate
`sin cos tan log exp sqrt log log10`	`(Floating a) => a -> a`	Usual math functions
`pi`	`(Floating a) => a`	3.141592653589793
`round truncate`	`(RealFrac a, Integral b) => a -> b`	Conversion to integer

Functions on characters

Some functions require `import Data.Char`.

Function	Type	Comment
`ord`	`Char -> Int`	Convert to integer ASCII value
`chr`	`Int -> Char`	Convert from ASCII value
`isPrint isSpace`	`Char -> Bool`	Test for printable or nonprintable character
`isAscii isControl`	`Char -> Bool`	Test for ASCII (?), control character
`isUpper isLower`	`Char -> Bool`	Test for capital or lowercase letter
`isAlpha isDigit`	`Char -> Bool`	Test for letter, test for decimal digit
`isAlphaNum`	`Char -> Bool`	Test for letter or digit

Functions on Booleans

Function	Type	Comment
`&& ||`	`Bool -> Bool -> Bool`	Infix **and**, infix **or**
`not`	`Bool -> Bool`	Prefix **not**

Typeclasses

A typeclass describes certain operations that a type can perform. A type may belong to multiple typeclasses. Some typeclasses are:

Typeclass	Operations	Comment
`Eq a => a -> a -> Bool`	`== /=`	Equality and inequality; arguments must have same type
`Ord a => a -> a -> Bool`	`< <= >= >`	For ordered values; arguments must have same type
`Num t => t`	Numeric operations	Describes numbers; not a subclass of `Ord`
`Integral`	Operations on integers	
`Floating`	Operations on floating-point numbers	
`Enum a => a -> a`	`succ pred`	Can be used in defining ranges
`Bounded a => a`	`minBound maxBound`	Provides type-specific constants
`Read a => String -> a`	`read`	Must specify desired type, e.g.: `read "5" :: Float`
`Show a => a -> String`	`show`	Convert almost anything to a string

Lists and Strings

Haskell also contains lists, tuples, and functions.

Lists are written using brackets and commas, e.g. `[1,2,4,8]`. All the elements of a list must be of the same type; the type of a list is denoted `[T]`, where *T* is the type of the elements. The list `[a]`, where *a* is a variable, denotes a list whose elements are all of type *a*.

The empty list is denoted by `[]`. Lists can be appended (concatenated) with the `++` operator.

A string is a list of characters, that is, it has type `[Char]`. For convenience, a string may be written with double quotes, e.g. `"Haskell"` is the same thing as `['H','a','s','k','e','l','l']`. Because strings are lists, they may be concatenated with the `++` operator.

Functions on lists (and therefore strings):
Note: Some functions require `import Data.List`.

Function	Type	Comment
`null`	`[a] -> Bool`	Test if a list is empty
`:`	`a -> [a] -> [a]`	(infix) Add an element to the front of the list
`++`	`[a] -> [a] -> [a]`	(infix) Append two lists
`!!`	`[a] -> Int -> a`	(infix) Return element Int of the list, counting from zero
`head`	`[a] -> a`	Return the first element of the list

`tail`	`[a] -> [a]`	Return the list with the first element removed
`last`	`[a] -> a`	Return the last element in the list
`init`	`[a] -> [a]`	Return the list with the last element removed
`reverse`	`[a] -> [a]`	Return the list with the elements in reverse order
`take`	`Int -> [a] -> [a]`	Return the first Int elements of the list
`drop`	`Int -> [a] -> [a]`	Return the list with the first Int elements removed
`nub`	`[a] -> [a]`	Return the list with all duplicate elements removed
`elem notElem`	`[a] -> a -> Bool`	Test for membership in the list
`length`	`[a] -> Int`	The number of elements in the list
`concat`	`[[a]] -> [a]`	Given a list of lists, concatenate all the lists into one list

There are various shortcuts to writing lists. Haskell uses a technique called *lazy evaluation:* no value is ever computed until it is needed. Lazy evaluation allows Haskell to support *infinite lists* (and other infinite data structures). Arithmetic over infinite lists is supported, but some operations must be avoided, for example, it is a bad idea to ask for the last element of an infinite list.

Here are some list notations, with brief explanations and examples:

Notation	Explanation	Example
`[a..b]`	The list of all values from *a* to *b*, inclusive	`[1..5] == [1,2,3,4,5]` `[2.. -2] == []`
`[a..]`	The list of all values equal to or larger than *a*	`[1..] == all positive integers`
`[a, b..c]`	The list of values starting from *a* and stepping by `(b-a)`, up to and possibly including *c*	`[1,3..10] == [1,3,5,7,9]` `[2,1.. -2] == [2,1,0,-1,-2]`
`[a, b..]`	The list of values starting from *a* and stepping by `(b-a)`	`[1, 3..] == all odd positive integers`
`[expression_involving_x | x <- list]`	The set of all values of the expression involving *x*, where *x* is drawn from the list (this is called a *list comprehension*)	`[x*x | x <- [1..]] == the list of squares of positive integers`
`[expression_involving_x_and_y | x <- list, y <- list]`	The set of all values of the expression involving *x* and *y*, where *x* is drawn from one list and *y* from the other (*y* varies faster)	`[[x,y] | x <- ['a'..'b'], y <- ['x'..'z']] == ["ax", "ay", "az", "bx", "by", "bz"]`
`[expression_involving_x | x <- list, condition_on_x]`	The set of all values of the expression involving *x*, where *x* is drawn from the list and	`[x*x | x <- [1..10], even x] == [4, 16,`

	meets the condition	`100]`
`[`*expression_involving_x_and_y* `|` *x* `<-` *list*`,` *condition_on_x*`,` *y* `<-` *list*`,` *condition_on_y*`]`	The set of all values of the expression involving *x* and *y*, where *x* is drawn from one list and meets one condition, while *y* is drawn from the other list and meets the other condition	`[x+y	x <- [1..5], even x, y <- [1..5], odd y] == [3, 5, 7, 5, 7, 9]`

As an example of the use of lazy evaluation, the infinite list `[x*x | x <- [1..]]` is the list of squares of positive integers. A useful expression involving this list is `take 10 [x*x | x <- [1..]]`.

Tuples

Tuples are written using parentheses and commas, e.g. `("John",24,True)`. The elements of a tuple may be of different types. Two tuples have the same type if they have the same number of elements and those elements have the same types in the same order.

There are very few operations defined on tuples. If a tuple has exactly two elements, the functions `fst` and `snd` return the first and second elements, respectively. In general, you access the elements of a tuple by pattern matching, as explained below.

Functions

Function definitions must be loaded from a file; they cannot be defined from the Haskell prompt.

In Haskell, a function is a "first-class object," able to be used the same way other types are used (e.g. passed as arguments to functions).

Functions are defined using the `=` operator. For example,

```
averageOf2 x y = (x + y) / 2
```

This is actually syntactic sugar for the following:

```
averageOf2 = \x y -> (x + y) / 2
```

The expression on the right of the equals sign indicates an (anonymous) function of two arguments, `x` and `y`, with the body of the function following the arrow; this function is "assigned to" the variable on the left of the equals sign. (The backslash in this expression is pronounced "lambda.") Anonymous functions are used frequently in Haskell, usually surrounded by parentheses.

Haskell supports function "sections." For example, given the above definition, the expression `(averageOf2 25)` denotes the function that returns the average of 25 and its (single) argument. As a further example, you could define a Boolean function `isNegative` as

```
isNegative = (< 0)
```

(The parentheses are necessary.) Other examples of function sections are `(2 *)`, `(2 /)`, `(/ 2)`, and `(1 -)`. However, `(- 1)` does not work, because '-' is considered to be the unary minus; use `(subtract 1)` instead.

Due to sectioning, all Haskell functions could be thought of as having a single argument; for example, a function with type `a -> b -> c -> d` may be considered to be a function that takes a value of type `a` as an argument and returns as result a new function of type `b -> c -> d`, which in turn takes a value of type `b` as an argument and returns a new function of type `c -> d`, which takes a value of type `c` as an argument and returns a result of type `d`. Thus, the implicit parenthesization is: `a -> b -> c -> d == a -> (b -> (c -> d))`

Here are some functions that take functions as arguments:

Function	Type	Comment and example
`map`	`(a -> b) -> [a] -> [b]`	Applies the function to all elements of the list: `map chr [104, 101, 108, 108, 111] == hello`
`filter`	`(a -> Bool) -> [a] -> [a]`	Returns the list of elements for which the function returns `True`: `filter odd [1..5] == [1, 3, 5]`
`iterate`	`(a -> a) -> a -> [a]`	returns the list `[x, f x, f f x, f f f x, ...]`: `take 5 (iterate (2*) 1) == [1, 2, 4, 8, 16]`
`foldl`	`(a -> b -> a) -> a -> [b] -> a`	`foldl f i x` starts with the value `i` and accumulates the value obtained by repeatedly applying `f` to the current value and the next element in the list: `foldl (-) 100 [1..3]` computes the value `(((100-1)-2)-3)` *Cannot be used on infinite lists!*
`foldl1`	`(a -> a -> a) -> [a] -> a`	Same as `foldl f (head x) (tail x)`
`foldr`	`(a -> b -> b) -> b -> [a] -> b`	`foldr` is like `foldl`, but works from the right end of the list.
`foldr1`	`(a -> a -> a) -> [a] -> a`	Same as `foldr1 f (last x) (init x)`
`flip`	`(a -> b -> c) -> b -> a -> c`	Return a function whose first two arguments are reversed: `flip elem "aeiou" 'y'`is the same as `elem 'y' "aeiou"`
`(.)`	`(b -> c) -> (a -> b) -> a -> c`	Compose two functions into a single function: `(f . g)` x is the same as `f (g x)`
`span`	`(a -> Bool) -> [a] -> ([a],[a])`	Break the list into a tuple of two lists: all those at the front of the list that satisfy the test, and the rest of the list: `span odd [3,1,4,1,6] == ([3, 1],[4, 1, 6])`
`break`	`(a -> Bool) -> [a] -> ([a],[a])`	Break the list into a tuple of two lists: all those at the front of the list that fail the test, and the rest of the list: `break even [3,1,4,1,6] == ([3, 1],[4, 1, 6])`

`takeWhile`	`(a -> Bool) -> [a] -> [a]`	Takes as many elements from the front of the list as satisfy the predicate.
`dropWhile`	`(a -> Bool) -> [a] -> [a]`	Removes as many elements from the front of the list as satisfy the predicate.
`zipWith`	`(a -> b -> c) -> [a] -> [b] -> [c]`	Takes a function of two parameters, and applies the function to corresponding elements of two lists: `zipWith (\x y -> 100 * x + y) [1, 2, 3, 4] [5, 6, 7] == [105,206,307]`
`all`	`(a -> Bool) -> [a] -> Bool`	Univeral quantification, ∀: Test if the predicate is true for all elements in the list.
`any`	`(a -> Bool) -> [a] -> Bool`	Existential quantification, ∃: Test if the predicate is true for at least one element in the list.
`$`	`(a -> b) -> a -> b`	Function application, but with lowest precedence. `f g h x` means `((f g) h) x`, but `f $ g $ h x` means `f(g(h x))`.

List comprehensions are equivalent to combinations of `map` and `filter` functions. Sometimes list comprehensions are more readable; sometimes the explicit combination of `map` and `filter` functions is more readable.

`Data.List` contains many additional higher-order functions for working with lists.

Maybe

Some functions may or may not return a suitable value; for example, `find (< 0) [1, 2, 3]` will fail to find a negative number in the list. The type of `find` is `(a -> Bool) -> [a] -> Maybe a`, where the result will be either `Just a` for some value *a*, or it will be `Nothing`.

Pattern matching

Parameter transmission is by *pattern matching;* thus, a function definition may consist of two or more equations with differing formal parameters. For example, the factorial function may be defined as:

```
fact 0 = 1
fact n = n * fact (n - 1)
```

Equations will be tried in order; the pattern `0` in the first equation for fact will match only a call with a zero argument, while the pattern `n` in the second equation will match anything.

Here are the allowable patterns:

Type of pattern	Examples	Comment
A variable	`x`	Will match anything
A constant	`5 'a' []`	Will match only that value

Wildcard	`_`	"Don't care"--will match anything, but the matched value can't be used
Tuples	`(x, y)`	Matches a pair--the component values can be referred to as `x` and `y`
Fixed-length lists	`[x]` `[x,_,y]` `[]`	With the bracket notation, you can only match lists of the given length
Variable-length lists	`(h:t)`	Matches a nonempty list whose head is `h` and whose tail is `t`
As-patterns	`p@(x,y,z)` `p@(h:t)`	When the pattern after the `@` matches something, the variable `p` matches the whole thing; e.g. if `p@(h:t)` matches `[1,2,3]`, then `h` matches `1`, `t` matches `[2,3]`, and `p` matches `[1,2,3]`
(n+k) patterns	`(m+1)`	Matches any value equal to or greater than `k` (`k` must be an integer); `m` is `k` less than the value matched

Pattern matching allows you to write your own functions to extract the fields of tuples. For example, if you have tuples of the form `("Mary", 'f', 38)`, you might write extraction functions as follows:

```
name (x, _, _) = x
gender (_, x, _) = x
age (_, _, x) = x
```

Expressions

`if` expressions

The `if` construct in Haskell is an expression, not a statement; it must return a value. Therefore, the `else` part is required. The syntax is:

```
if Bool_expression then result else result
```

`case` expressions

Patterns may be used in *case expressions,* with the syntax:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

The first printing character following the word `of` in a case statement defines the starting column for that group of cases; all additional cases must begin in the same column.

`let` expressions

A *let expression* defines local variables to be used in a subsequent expression:

```
let var = value
```

```
    var = value
    ... in result
```

The values may be functions. The variables must all begin in the same column, or be separated by semicolons.

Styles of writing functions

A function consisting of multiple equations may have *guards* on each equation, as for example:

```
fact n
    | n == 0    = 1
    | otherwise = n * fact (n - 1)
```

Another way to write this function is:

```
fact n = case n of
           0 -> 1
           n -> n * fact (n - 1)
```

Or as:

```
fact 0 = 1
fact n = n * fact (n - 1)
```

Or as:

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

You can introduce new local variables or functions with an expression of the form

```
let declarations in expression
```

for example:

```
fact n
    | n==0      = 1
    | otherwise = let m = n - 1 in n * fact m
```

Another way to introduce local variables or functions is:

```
function_definition where declarations
```

as for example in:

```
fact n
    | n==0      = 1
    | otherwise = n * fact m
    where m = n - 1
```

Whereas `let` is an expression and may be used wherever an expression may be used, `where` is part of the syntax of a function declaration, and can only be used in

this way. The scope of variables declared by `let` is the single case in which it occurs; the scope of `where` is the entire equation.

Haskell is a strongly-typed language, but you seldom need to tell Haskell what type your variables and functions are, because Haskell can almost always figure it out. It is a good idea to specify the types of functions, however, both for documentation, and so that Haskell can inform you if a function doesn't have the type you intended. The operator `::` means "has type," and can be used as follows:

```
fact :: Int -> Int
```

Layouts

Indentation is sometimes important in Haskell.

An equation starts on a new line. Successive lines must be indented; when Haskell finds a line beginning in the same (or earlier) column as the start of the last equation, it takes this as the start of the next equation. Suggested indentation: start every equation in column 1, and indent all remaining lines of the equation. See the various definitions of `fact` above for examples.

The first declaration following `let` or `where` defines the starting column for that group of declarations; all additional declarations must begin in the same column.

```
latinize :: [Char] -> [Char]
latinize word = (snd pair) ++ (fst pair) ++ "ay"
    where vowel = flip elem "aeiou"
          consonant x = isLower x && not (vowel x)
          pair = span consonant word
```

Input/Output

Input and output are side effects. Moreover, a request for input is not referentially transparent: It will not always return the same value. For these reasons, I/O does not belong in a purely functional language. Since I/O is necessary for practical reasons, Haskell isolates I/O as much as possible.

A Haskell program contains a definition of the name `main` as a value of type `IO(`*`something`*`)`, which is one *I/O action*. A sequence of I/O actions can be grouped with `do`, similar to the way statements in other languages can be grouped into a single statement with braces, `{}`. The I/O actions in a `do`-group are executed sequentially, very much like programming in an imperative language. A `do`-group may be bounded by enclosing it in parentheses.

I/O actions for user interaction

```
putStrLn :: String -> IO ()
```
Print a string to standard output, followed by a newline.

```
putStr :: String -> IO ()
```

Print a string to standard output, not followed by a newline.

`putChar :: Char -> IO ()`
Print a single character to standard output.

`hFlush stdout`
Ensure that all previous `putXxx` statements have been performed.

`print :: Show a => a -> IO ()`
Print any showable value to standard output, followed by a newline. Puts quotes around Char and String values.

`getChar :: IO Char`
Read a character from standard input and put it in an I/O action. (Within a `do`, you can use `c <- getChar`.)

`getLine :: IO String`
Read a string from standard input, discarding the newline, and put it in an I/O action. (Within a `do`, you can use `ln <- getLine`.)

`interact :: (String -> String) -> IO ()`
Repeatedly reads a line from standard input, applies the function, and prints the result to standard output.

`getContents :: IO String`
Reads all input from standard input, but does so lazily. That is, the input is only read as needed; the result is a lazy list of characters. (This takes some getting used to.)

I/O actions for files

`readFile :: FilePath -> IO String`
Reads in the contents of a file as a single string. `FilePath` is just a string.

`writeFile :: FilePath -> String -> IO ()`
Writes a string to a file. `FilePath` is just a string.

`appendFile :: FilePath -> String -> IO ()`
Writes a string to a file. `FilePath` is just a string.

`openFile :: FilePath -> IOMode -> IO Handle`
IOMode is one of: `ReadMode`, `WriteMode`, `AppendMode`, `ReadWriteMode`.
An IO Handle is like a stream in other languages.
To use `openFile` and the following functions that use handles, `import System.IO`.

```
hPutStrLn :: Handle -> String -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutChar :: Handle -> Char -> IO ()
hPrint :: Show a => Handle -> a -> IO ()
hFlush :: Handle -> IO ()
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String
hClose :: Handle -> IO
```
As above, but for file handles rather than standard input and standard output.

`getCurrentDirectory :: IO FilePath`
Returns the current directory; in module `System.Directory`.

General I/O actions

`return :: Monad m => a -> m a`
: `return` is a **function** that, given any value, returns an I/O action "containing" that value. It does **not** cause the enclosing function to return.

`name` `<-` *`IO_action`*
: Syntax (not a function) that extracts a value from an I/O action, for example, `line <- getLine.`
`name` `<- return` *`value`* acts like a `let` binding, and can be used where an I/O action is required.

`when :: Monad m => Bool -> m () -> m ()`
: `when` returns the I/O action `m()` if the Bool is true, otherwise it returns the "do-nothing" I/O action `return()`. [Defined in `Control.Monad`.]

`sequence :: Monad m => [m a] -> m [a]`
: `sequence` collects a list of I/O actions into a single I/O action, for example, `sequence(map print [1,2,3,4,5])`.

`mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
: `mapM` *`function list`* is equivalent to `sequence(map` *`function list`*`)`.

`mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`
: Like `mapM`, but returns the "do-nothing" I/O action `return()`.

`forM :: Monad m => [a] -> (a -> m b) -> m [b]`
: `forM` *`list function`* is the same as `mapM` *`function list`*.

`forever :: Monad m => m a -> m b`
: Repeats the I/O action forever.

`catch :: IO a -> (IOError -> IO a) -> IO a`
: Evaluates the first argument; if an exception occurs, evaluates the second argument.

Any function resulting in an IO action may be used within an I/O action. All I/O must be performed from `main` or from functions of type `IO(`*`something`*`)` called, directly or indirectly, from `main`. A non-IO function cannot call an IO function.

`let` bindings may be used in an IO function, and the scope is the remainder of the function. Indentation is significant, and the word `in` (normally used with `let`) must be omitted.

Not discussed

Haskell is a rich language, and a number of important topics have not been discussed in this paper. Among these are irrefutable patterns, user-defined data types and type synonyms, polymorphism, overloading, contexts, and dictionaries.

A final example

The following definition of the quicksort algorithm shows just how nice Haskell can be (at least sometimes).

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (pivot:xs) =
     quicksort [x | x <- xs, x < pivot] ++
     [pivot] ++
     quicksort [x | x <- xs, x >= pivot]
```

References

Learn You a Haskell for Great Good!, http://learnyouahaskell.com/.