A Tutorial Introduction to Haskell[1]

Quentin Miller

Somerville College
Oxford
2022

[1]adapted from "A Tutorial Introduction to Orwell" by Philip Wadler and Quentin Miller.

1 Introduction

To begin programming in Haskell, you'll need to install it on your computer. Instructions for doing so on Windows, MacOS, and Linux systems are at

```
https://www.haskell.org/downloads/
```

Alternatively, if you have the Homebrew package installer on MacOS or Linux, you can install Haskell with the command

```
brew install ghc
```

(Don't worry if the package installer for your system provides an outdated version; it should be fine for this tutorial.) If you have any trouble installing Haskell on your own computer, don't hesitate to email me at `<quentin.miller@some.ox.ac.uk>` for help.

The basic installation will set up a compiler (the *Glasgow Haskell Compiler* or *GHC*) and an interpreter (the *GHC interpreter* or *GHCi*), which are invoked from a command line in a terminal window with the commands `ghc` and `ghci` respectively. In this tutorial, only the interpreter is needed. Open a new terminal window and type `ghci` followed by a carriage return, in response to the prompt. You should see something like this:

```
quentin$ ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/ :? for help
Prelude>
```

You are now in an interactive GHCi session. You can exit and return to a terminal prompt by typing `:quit` (note that it begins with a colon) followed by carriage return. If you haven't managed to get to this point, email me for help. Otherwise, read on . . . .

□

2 Sessions

Having installed Haskell on your computer, you should be able to execute the command `ghci` in a terminal window, and see this displayed:

```
quentin$ ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

You are now in a GHCi *session*: the `Prelude>` at the end is a *prompt* (the significance of "Prelude" will be explained later), and if you type an arithmetic expression and press *return*, the interpreter will evaluate the expression, display the result, and present a new prompt:

```
Prelude> 2+1
3
Prelude> 3
3
Prelude> 2+3*2
8
Prelude> (2+3)*2
10
Prelude> (2+3)/2
2.5
Prelude> 5 `div` 2
2
Prelude> 5 `mod` 2
1
```

Notes:

- Multiplication is denoted by an asterisk (`*`).
- Without parentheses, the usual arithmetic rules for operator precedence are assumed; e.g. 2+3*2 is taken to mean 2+(3*2).
- Integer division is performed by the operators `` `div` `` and `` `mod` `` (enclosed in back-quotes), which give the quotient and remainder respectively.

As well as individual numbers, Haskell can work with lists of numbers in square brackets, separated by commas:

```
Prelude> [3, 2, 2, 4]
[3,2,2,4]
```

A series of consecutive numbers is given by a pair of dots separating the first and last elements:

```
Prelude> [3..11]
[3,4,5,6,7,8,9,10,11]
```

If you give the second element of the series, the same interval will be used for the rest of the series:

```
Prelude> [3,5..11]
[3,5,7,9,11]
Prelude> [3, 4.5 .. 9]
[3.0,4.5,6.0,7.5,9.0]
Prelude> [11,10..3]
[11,10,9,8,7,6,5,4,3]
```

Haskell can add or multiply together all the elements of a list:

```
Prelude> [1..5]
[1,2,3,4,5]
Prelude> sum [1,2,3,4,5]
15
Prelude> sum [1..5]
15
Prelude> product [5..7]
210
```

⋆ **Exercise 1:**
What is the value of `[3..4]`? `[3..3]`? `[3..2]`?

⋆ **Exercise 2:**
Use lists (with the `..` notation) to calculate the factorial values 3!, 5!, and 10!.

⋆ **Exercise 3:**
The expressions you gave in the previous exercise should have the same pattern as one another, differing only in the 3, 5, and 10. If you substitute 1 or 0 for the 3, 5, and 10, do you get the correct values for 1! and 0!? (Recall that by definition, $1! = 1$ and $0! = 1$.)

As well as typing an expression for evaluation after the `Prelude>` prompt, you can type a command. All commands begin with a colon, and they can be given in full or abbreviated to the first letter. The first command you'll need to know about is the *quit* command (`:quit` or `:q`) that will end the GHCi session.

□

3 Scripts

The prompt in our interactive sessions is `Prelude>` — this refers to a file called *Prelude.hs* that contains useful definitions such as such as `sum` and `product` that we can then use in the session. Such a file of Haskell definitions is called a *script*. *Prelude.hs* (often referred to as the *standard prelude*) is loaded automatically when we start the Haskell interpreter, but we can load additional scripts containing some definitions of our own. The filename extension *.hs* stands for *Haskell Script*.

A script file contains plain text, so we may create it using an ordinary text editor. Stop the session (with the `:q` command), and start up an editor in your terminal window. (Any plain text editor will do; for example `emacs` or `vi` which run directly in the terminal window, or `notepad` or `TextEdit` which run in a separate window.) Create a file with the name `script1.hs`, and type the following definitions into it:

```
square x = x*x
cube x = x*x*x
percent n x = (x*n) `div` 100
```

After starting the interpreter (with `ghci`) we can load the definitions from our new script into the session by typing `:load script1.hs` to which the interpreter should respond:

```
Prelude> :load script1.hs
[1 of 1] Compiling Main             ( script1.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

The new `*Main>` prompt indicates that this session includes some user-defined functions as well as those defined in the standard prelude. We can now evaluate expressions containing our new definitions:

```
*Main> square 3
9
*Main> cube 3
27
*Main> square (cube 3)
729
*Main> percent 12 64
7
*Main> percent 12 (square 8)
7
```

Notes:

- Names of functions and their parameters **must not** begin with capital letters. (The reason will be explained later.)
- Haskell's notation for functions differs from that of most other programming languages. `f x` means "function `f` applied to argument `x`" (in the same way that "$\sin x$" means the sine function applied to x in conventional mathematical notation). In a language such as C++ or Java this would be written `f(x)`.

- Parentheses can however be placed around any Haskell expression or any of its constituent sub-expressions; thus we could write `f(x)` but it would mean the same thing as `f x` or `(f) x` or `(f x)` or `(((f))(x))`.

Without parentheses, function application always takes precedence over infix operators (i.e. operators such as `+` and `*` that appear between their operands.) Thus, `f x + y` is the same as `(f x) + y`. Adding extra space has no effect on precedence (but it might make an expression more human-readable):

```
*Main> square 3+5
14
*Main> square 3 + 5
14
*Main> (square 3) + 5
14
*Main> square (3 + 5)
64
```

A Haskell expression of the form `a b c` is taken to mean the function `a` applied to arguments `b` and `c`. So if we want to take the square of the cube of 3, we must write this as `square (cube 3)`. Without parentheses, it would be equivalent to `(square cube) 3` — where `cube` rather than `(cube 3)` is the argument of `square` — which is not what we want.

⋆ Exercise 4:

Add to your script file the definition of a new function `factorial` which calculates the factorial of a number. After changing and saving the script file, load it into a session to test it. If the editor is running in the same terminal window as the interpreter, you will need to exit the editor, invoke *ghci*, and execute the `:load` command. If the editor is running in another window you don't need to stop and start GHCi; just type `:reload` or `:r` (remembering to save the edited script first).

Test your factorial function on several values, including 0 and 1.

□

4 Functions

Haskell is called a *functional* language because it can manipulate functions in the same way it manipulates numerical and other values. The standard prelude contains a `map` function which, when applied to another function and a list, applies that function to every element in the list:

```
*Main> map square [1,5,10]
[1,25,100]
*Main> map square [0..9]
[0,1,4,9,16,25,36,49,64,81]
```

The list needn't be given explicitly within square brackets; it might be the result of another calculation:

```
*Main> map cube (map square [1,5,10])
[1,15625,1000000]
*Main> map square (map square [0..9])
[0,1,16,81,256,625,1296,2401,4096,6561]
```

A function, in Haskell, is any expression that can be applied to another expression. Some functions have names (`square`, `map`, etc.), but a nameless function can be created. One way of creating such an *anonymous function* is with the *composition* operator. Mathematicians usually denote this with a small circle ($\circ$); in Haskell we use a dot (.). When applied to two functions, it *composes* them into a single "pipelined" function which uses the output of one as the input to the other. Specifically, for any functions f and g, the effect of applying their composition $(f.g)$ to an argument is defined by

$$(f.g)\ x = f\ (g\ x)$$

Thus, $f.g$ is a function like any other; the left-hand side of the above equation shows that it can be applied to an expression, and the right-hand side shows what the effect of such an application is. We can rewrite our earlier example of `map cube (map square [1,5,10])` as the mapping of a single composed function, rather than two separate mappings:

```
*Main> map (cube.square) [1,5,10]
[1,15625,1000000]
```

Anonymous functions can also be created by *partial application*. Our script file contained the definition

```
        percent n x  =  (x*n) `div` 100
```

which says that "`percent`, when applied to any n and x is equal to `(`x`*`n`) `div` 100`". Another way of saying that is "`percent`, when applied to any n, is an expression that when applied to any x is equal to `(`x`*`n`) `div` 100`". Remembering that any expression that can be applied to another expression is by definition a function, that means that `percent 25` is an anonymous function that yields 25% of any value to which it is applied. Here is a list of numbers, all scaled by 25% (rounded down):

```
        *Main> [0,10..100]
        [0,10,20,30,40,50,60,70,80,90,100]
        *Main> map (percent 25) [0,10..100]
        [0,2,5,7,10,12,15,17,20,22,25]
```

(The parentheses in the above example are necessary; we want `percent` — not `map` — to be applied to 25.) As `percent 25` is a function, it can be composed with another function:

```
        *Main> map square [0..10]
        [0,1,4,9,16,25,36,49,64,81,100]
        *Main> map (percent 25 . square) [0..10]
        [0,0,1,2,4,6,9,12,16,20,25]
```

A special notation allows us to partially apply infix operators. A *section* is the partial application of an operator, enclosed in parentheses. An expression of the form x @ y, where @ is any infix operator (such as `+`, `*`, or `` `div` ``) can be given in any of these ways, all of which represent the same value:

x @ y
`(`@`)` x y
`(`x @`)` y
`(`@ y`)` x

So for any x, `(+1)` x is equivalent to x`+1`, hence we can use `(+1)` as a function that increments its argument:

```
*Main> [0..9]
[0,1,2,3,4,5,6,7,8,9]
*Main> map (+1) [0..9]
[1,2,3,4,5,6,7,8,9,10]
```

There is one exception to these equivalences: `-` can be used as an infix or a prefix operator, so sections of `-` are interpreted as follows:

`(-)`	binary `-` (subtraction)
`(-` a`)`	unary `-` (negation)
`(`a `-)`	binary `-` (subtraction)

Haskell allows any function of two parameters to be used as an infix operator, by enclosing it in back-quotes (`` ` ``). You've seen this with ``x `div` y``, which is equivalent to `div x y`. The `map` function has two parameters, so you can use it as an infix operator:

```
*Main> map (+1) [0..9]
[1,2,3,4,5,6,7,8,9,10]
*Main> (+1) `map` [0..9]
[1,2,3,4,5,6,7,8,9,10]
```

When used in a section, this can partially apply a function to its second rather than its first parameter. For example, we've seen that `percent 25` is a function that yields 25% of the value to which it is applied. Now observe that

$$\texttt{percent}\; n\; x \;=\; n\; \texttt{`percent`}\; x \;=\; (\texttt{`percent`}\; x)\; n$$

so ``(`percent` 500)`` is a function that when applied to n yields n% of 500:

```
*Main> map (`percent` 50) [0..9]
[0,0,1,1,2,2,3,3,4,4]
*Main> map (`percent` 500) [0..9]
[0,5,10,15,20,25,30,35,40,45]
```

⋆ Exercise 5:

Write an anonymous function that doubles its argument and then squares the result. Test it by mapping it over the list `[0..9]`.

⋆ Exercise 6:

(Challenge!) As well as lists of numbers, we can have lists of lists; for example, `[[1..5], [10], [22,23,24]]` is a three-element list, each of whose elements is a list of numbers. Fill in the "?" in the expression below so that each number in each list is doubled and then squared:

```
*Main> map ? [[1..5], [10], [22,23,24]]
[[4,16,36,64,100], [400], [1936,2116,2304]]
```

□

5 Function Definitions

Functions can be defined using recursion and conditionals. For example, one may define a function to raise x to the n^{th} power as follows:

```
power x n  =  if n==0 then 1
              else if n>0 then x * power x (n-1)
              else error "negative power"
```

(If you type this definition into a script to try it, you must indent the lines as they are shown above. The reason for this is explained below.)

Notes:

- `power x n` is the application of the `power` function to two arguments `x` and `n`.
- `==` is the operator that tests two expressions for equality. It is written this way to distinguish it from the `=` which separates the two sides of an equation in a function definition.
- In Haskell, conditional expressions have the familiar form: `if` e_1 `then` e_2 `else` e_3
- The `error` function is built into Haskell. It allows you to generate an error message of your choice.

Another useful language feature which Haskell provides is a "guarded expression". The `sign` function below takes an argument `x`. The remaining parts of the function are protected by "boolean guards". One of the guards (`x>0`, `x==0` and `x<0`) will evaluate to `True`. The expression associated with the `True` guard is produced as the result of the function.

```
sign x | x>0   =  1
       | x==0  =  0
       | x<0   =  -1
```

The right-hand side of an equation may also be followed by a `where` clause, which defines the value of one or more names. For example, this definition gives the number of solutions of a polynomial of the form $ax^2 + bx + c = 0$:

```
numberOfRoots a b c | disc > 0   =  2
                    | disc == 0  =  1
                    | disc < 0   =  0
    where disc = b*b - 4*a*c
```

Defining `disc` as a local value not only makes the definition easier to read, it improves efficiency by ensuring that it is calculated only once.

A `where` clause in Haskell may define constants (as above) or functions of any number of arguments.

In many languages, extra brackets and symbols (such as `{`, `}` and `;`) are needed to indicate the extent of an expression. In Haskell, this information is indicated by indentation: definitions at the same level (*i.e.* at the top level or within the same `where` clause) must begin in the same column. For example:

```
inc x = x + 1
dec x = x - 1
```

is interpreted as two function definitions, while

```
inc x = x + 1
 dec x = x - 1
```

would be interpreted as the single definition

```
inc x = x + 1 dec x = x - 1
```

which is meaningless and will generate an error.

Thus, long definitions may be broken across lines simply by indenting all but the first line of the definition:

```
bigAndSmall = (aBigNumber, aSmallerNumber)
        where aBigNumber = 1 + 1 + 1 + 1 + 1 + 1 +
                           1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +
                           1 + 1 + 1 + 1 +
                           1 + 1
              aSmallerNumber = 1 + 1 + 1
```

⋆ Exercise 7:

Define a function `fib` such that `fib n` gives the n^{th} Fibonacci number. These numbers are defined by:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{k+2} &= F_k + F_{k+1} \end{aligned}$$

Be careful how you use parentheses in your definition: function application in Haskell binds more tightly than infix operators. For example,

```
f x + 3
```

is the sum of $(f\ x)$ and 3, while

```
f (x + 3)
```

is the function f applied to $(x + 3)$.

□

6 Basic Data Types

Life would be dull if we could write functions only on numbers. In addition to numbers, Haskell provides operations on booleans, characters, lists, tuples, and functions, as well as on other types the user may care to define.

Integers are written in standard decimal notation, e.g., `42`. Standard operations on integers are `+`, `-`, `*`, `^`, `` `div` ``, `` `mod` ``, and prefix `-`. The boolean values are written `True` and `False`. Boolean values are given by the comparison operators: `==`, `/=` (not equal), `<`, `>`, `<=`, and `>=`. Boolean values may also be combined using `not`, `&&`, and `||` (*not*, *and*, and *or* respectively).

□

7 Lists

Lists are written enclosed in square brackets with commas separating elements, for example `[1, 2, 3]`. The empty list is written `[ ]`, and a list containing just the number four is written `[4]`. (Spaces between elements, or in an empty list, are optional.) One may have lists of any type, for example `[True,False]` is a list of booleans and the nested list `[[1,2,3], [], [1]]` is a list of lists of integers. All elements of a list must have the same type.

The operator `:` (pronounced *cons*) adds an element to the front of a list. For example, `1:[2,3]` is `[1,2,3]` and `4:[]` is `[4]`. In fact, `[1,2,3]` is just an abbreviation for `1:2:3:[]` (which is in turn an abbreviation for `1:(2:(3:[]))`, since `:` is right-associative).

When a list is created by prepending element `x` to list `xs` by `x:xs`, we call `x` the *head* of the list and `xs` the *tail* of the list.

Other operations on lists are `++` (*append*), `length`, and `!!` (*index*). And as we saw earlier, `[m..n]` is evaluated as the list of numbers from m up to n, while `[m, n..o]` yields the list of numbers from m up to o going by steps of $(n - m)$. If $m > n$, then the sequence will descend to o. For example:

```
*Main> [1,2] ++ [3,4]
[1,2,3,4]
*Main> length [0,1,2]
3
*Main> [10,11,12]!!1
11
*Main> [3..5]
[3,4,5]
*Main> [-8 .. -8]
[-8]
*Main> [5..3]
[]
*Main> [5,10..33]
[5,10,15,20,25,30]
```

Notes:

- In the example of `[-8 .. -8]`, there must be a space between the `..` and the `-` to prevent them from being interpreted as a single symbol.
- Indexing begins at zero.

One can define new functions on lists using recursion and patterns on the left-hand side containing `[ ]` and `:`. The following function squares every element of a list (assuming we still have our `square` function from earlier):

```
squares []     = []
squares (x:xs) = square x : squares xs
```

Note:

- All lists are of the form `[ ]` or the form `(x:xs)` — any non-empty list has a first element and a (possibly empty) tail — thus `squares` is defined for all possible integer lists. (Mathematicians call this a *total* function.)

★ **Exercise 8:**
Write a recursive function `listinsert` which takes a value and a sorted list, and yields the list with the value inserted into it so that the result is still sorted. `listinsert` should be total; thus you will need an equation for each form of list:

```
listinsert t []     = ...
listinsert t (x:xs) = ...
```

Hint: In the second equation, if $t \leq x$, what will the first element of the result be, and what will the remainder of the result be? What if $t > x$? Use an "if" clause to distinguish between these cases.

★ **Exercise 9:**
Using `listinsert`, write a recursive function `listsort` which sorts a list. Make sure that `listsort` is total.

□

8 Characters and Strings

Characters are written enclosed in single quotes, such as `'a'`. The function `ord` converts a character to the integer corresponding to its ASCII code, and `chr` is the inverse. For example, `ord 'a'` is `97`, and `chr 97` is `'a'`.

The `ord` and `chr` functions are in a script called `Char`; if you want to use these functions you need to ensure that the `Char` script is loaded. You can do this by putting the line

```
import Data.Char
```

as the first line of your script file.

A string is just a list of characters. For example, `"abc"` is equivalent to `['a', 'b', 'c']`. One may manipulate strings in the same way as lists; for example, `"Hi " ++ "there!"` is evaluated as `"Hi there!"`.

⋆ **Exercise 10:**
Characters have a well-defined ordering. If you have written the `listsort` function from the above exercise, try applying it to some strings.

⋆ **Exercise 11:**
Define a function `nextlet` which takes a letter of the alphabet and yields the letter coming immediately after it. Assume that `'a'` follows `'z'`, and `'A'` follows `'Z'`. **Hint:** Remember that if you want to use the `ord` and `chr` functions, the line `import Data.Char` must appear at the top of your script.

⋆ **Exercise 12:**
Define a function `digitval` which converts a digit character to its corresponding numerical value.

□

9 Tuples

All elements of a list must have the same type. Collections of two or more values of (possibly) different types may be expressed as tuples; for example `(1, 2, 'a')` is a triple comprising an integer, another integer, and a character.

The operation `zip` converts two lists to a list of pairs; for example `zip [1,2,3] "abc"` is evaluated as `[(1,'a'), (2,'b'), (3,'c')]`. The result list will have the same length as the shorter of the two input lists. Thus, `zip "hello" [4,2]` is evaluated as `[('h',4), ('e',2)]`.

⋆ **Exercise 13:**
The order operators `<` and `>` are defined for tuples. Experiment with these to see how tuples are ordered.

⋆ **Exercise 14:**
Suppose a date is represented by a triple of integers $(day, month, year)$. Using the knowledge gained from the previous exercise, define a function `age` that takes two dates, the first being the birth date of some individual P and the second the current date, and gives the age of P as a whole number of years. **Hint**: Your function can use tuple patterns on the left-hand side to extract the components of the dates:

```
age (d,m,y) (dc,mc,yc) = ...
```

⋆ **Exercise 15:**
Write a function that converts a date given as a triple of integers (*e.g.* `(15,4,2022)`) to a string representation (`"April 15th, 2022"`). **Hint:** the built-in function `show` will convert a number to its string representation.

□

10 Comprehensions

The *list comprehension* notation makes it easy to describe certain common operations on lists. This notation is very similar to set comprehensions from set theory. It will be explained first by example; then a more precise definition will be given.

The function that squares every element of a list (seen previously) can be defined as:

```
squares xs  =  [x*x | x<-xs]
```

Similarly, a function that squares only the odd elements of a list and discards the rest can be defined by:

```
oddsquares xs  =  [x*x | x<-xs, x`mod`2 == 1]
```

Vector addition (adding corresponding elements of two lists) can be defined using comprehension and the `zip` function on pairs:

```
vecadd xs ys  =  [x+y | (x,y) <- zip xs ys]
```

The cartesian product of two lists (a list of all pairs with one member from each list) can be defined by:

```
cp xs ys  =  [(x,y) | x<-xs, y<-ys]
```

For example, `cp [1,2] "abc"` yields the list

```
[(1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c')]
```

Note that the second variable changes more rapidly then the first.

The final example is a functional version of Quicksort:

```
qsort []      =  []
qsort (x:xs)  =  qsort [u | u<-xs, u<x] ++
                 [x] ++
                 qsort [u | u<-xs, u>=x]
```

`qsort` works as by considering the two possible forms of a list (empty and non-empty), and defining what a sorted version of each form is:

- A sorted version of an empty list is the empty list.
- A sorted version of a list consisting of an element `x` and subsequent elements `xs` is a sorted list of those elements of `xs` which are less than `x`, followed by `x`, followed by a sorted list of those elements of `xs` which are greater than `x`.

Here is a precise, if compact, definition of list comprehensions. A comprehension has the form `[`e `|` q_1`,` ... `,` q_n`]`, where e is an expression (of type t), and $q_1 \dots q_n$ are qualifiers; the comprehension has type $[t]$. Each qualifier q_i is either a generator or an expression of type boolean. Each generator has the form p_i `<-` e_i, where p_i is a pattern (of type t_i) and e_i is an expression (of type *list of* t_i). The pattern p_i may contain variables, which are local variables whose scope is the initial expression e and the qualifiers $q_{(i+1)} \dots q_n$.

⋆ **Exercise 16:**
Using a list comprehension and the `sum` function, define a function for counting the number of negative integers in a list.

⋆ **Exercise 17:**
Define x^n using a list comprehension. **Hint:** Don't use recursion; think about an informal definition of what x^n means.

⋆ **Exercise 18:**
Define a function which lists the divisors of a positive integer.

□

11 Lazy Evaluation

Say that we wish to find the first perfect number. (A number is *perfect* if the sum of its factors, including 1 but not including the number itself, is equal to the number.) If we know that the first perfect number is less than 1000 then we could write:

```
factors n     =  [i | i <- [1..n-1], n`mod`i == 0]
perfect n     =  sum (factors n) == n
firstperfect  =  head [n | n <- [1..1000], perfect n]
```

(The function `head` gives the first element of a non-empty list.) This program is correct, but there are two difficulties. First, the program appears to do rather more work than necessary, since it appears to find a list of all perfect numbers up to 1000, and then throw away all but the first (which is 6). Second, it is annoying to have to give an arbitrary limit, such as 1000.

These problems are solved by Haskell's evaluation strategy, which is called *lazy evaluation*. In lazy evaluation, only those parts of the program necessary to calculate the answer are evaluated. This means that only the first element of the list will be calculated by the program above, so that it in fact does no extra work. Further, it means that one is allowed to create infinite structures, which are expanded in memory only as needed. For example, `[1..]` yields the infinite list $[1, 2, 3, \ldots]$.

```
*Main> [1..]
[1, 2, 3, 4, 5, 6, 7, 8Interrupted.
```

The interpreter would have printed numbers indefinitely, had the user not interrupted the calculation by pressing CTRL-C. The calculation could not have completed in finite time.

Now look what happens when an expression requires only a finite part of an infinite list:

```
*Main> head [1..]
1
*Main> [2,4..]
[2, 4, 6, 8, 10, 12, 14, 16Interrupted.
*Main> [2,4..] !! 3
8
```

As soon as enough elements of the list have been calculated so that the required one is available, no further elements of the list are generated (unless they become needed later).

We may define the infinite list of all perfect numbers by writing:

```
perfects  =  [n | n<-[1..], perfect n]
```

Given this script, we can have the following session:

```
*Main> head perfects
6
*Main> perfects
[6,28,496Interrupted.
```

The first term, `head perfects`, causes the first perfect number to be printed. The second term, `perfects`, causes the infinite list of perfect numbers to be printed. The interpreter will continue searching for the next element of this list forever, or until an interrupt is typed.

Infinite lists may also be created using *where* clauses. For example:

```
dither  =  [yesno, noyes]
           where  yesno = "YES" : noyes
                  noyes = "NO" : yesno
```

defines `dither` to be a list containing two infinite lists: `(["YES", "NO", "YES", ...], ["NO", "YES", "NO", ...])`. If we try to print the whole of `dither` we will never get past the first list; however if we selectively print the second list then none of the first list will need to be calculated, and we can see as much of the second list as we please:

```
*Main> dither
[["YES", "NO", "YES", "NO", "YES", "NO"Interrupted.
*Main> dither!!0
["YES", "NO", "YES", "NO", "YES", "NO"Interrupted.
*Main> dither!!1
["NO", "YES", "NO", "YES", "NO", "YES"Interrupted.
```

Another property of lazy evaluation is revealed by the following script:

```
k x y  =  x
loop   =  loop
```

What is the value of `k 42 loop`? In many languages, the answer is that the program enters an infinite loop. But in a language with lazy evaluation, such as Haskell, the answer is 42.

★ **Exercise 19:**
Find the 100th prime number. **Hint:** Look at the above definitions for finding perfect numbers. Now, define a function which tests whether a number is prime; then define a list of all prime numbers; then use `!!` to select the 100th element.

★ **Exercise 20:**
(Challenge!) Define a list of all pairs (x, y) of nonnegative integers. Be careful to ensure that your answer is not restricted to cases where $y = 0$, or cases where $x \leq y$.

★ **Exercise 21:**
(Challenge!) A *pythagorean triple* is a triple (x, y, z) of positive natural numbers such that $x^2 + y^2 = z^2$. Define the list of all pythagorean triples. The list shouldn't include triples that are just rearrangements of other triples in the list; for example it shouldn't include both $(3, 4, 5)$ and $(5, 3, 4)$. A bigger challenge would be to exclude triples that are scalings of other triples in the list; for example not including both $(3, 4, 5)$ and $(6, 8, 10)$.

□

12 Higher-Order Functions

One important property of functional languages is that functions are values, just as numbers or lists are. Thus, a function may be an argument of a function, the result of a function, an element in a list, and so on. For example, the function

```
map f xs  =  [f x | x<-xs]
```

takes two arguments, a function `f` and a list `xs`, and applies `f` to every element of `xs`.

(Note that `map`, as well as `foldr`, `sum`, and `product` (below) are already defined in the standard prelude.)

A function may also be "partially applied" by giving only some of its arguments. For example, here is yet another way of defining the function that squares every element of a list:

```
squares  =  map square
```

A Haskell definition is in the form of an equation; *i.e.* a declaration that two things are equal. In mathematics, we may always substitute an expression for something else which it is equal to. Therefore given the above expression, the interpreter can evaluate `squares [1,2,3]` by replacing `squares` by `map square`, yielding `map square [1,2,3]`. This is an instance of `map f x`, which by our definition can be replaced by `[f x|x<-xs]`, yielding `[square x|x<-[1,2,3]]`. According to the definition of `square` given earlier, `square x` can be replaced by `x*x`, yielding `[x*x|x<-[1,2,3]]`. Thus the final result is calculated to be [1,4,9].

This style of programming can be quite powerful. For example, the following function defines a common pattern of computation:

```
foldr f a []      =  a
foldr f a (x:xs)  =  f x (foldr f a xs)
```

Functions to find the sum and product of all the elements of a list can be defined by:

```
sum      =  foldr (+) 0
product  =  foldr (*) 1
```

Note:

- To pass an infix operator such as `+` or `*` as an argument to a function, it must be written as `(+)` or `(*)`. (Recall the explanation of *sections*, above.)

Informally, given a binary operator $\oplus$ and an object e, $foldr\ (\oplus)\ e\ xs$ replaces each `:` in xs with $\oplus$ and the `[]` with e. Thus, it converts

$$x_0 : x_1 : x_2 : \ldots [\,]$$

to

$$x_0 \oplus x_1 \oplus x_2 \oplus \ldots e$$

With that in mind, let's have another look at how `sum` works:

$$\begin{array}{lcll} sum\ [x_0, x_1, x_2, x_3, \ldots x_n] & = & sum & (x_0 : x_1 : x_2 : x_3 : \ldots : x_n : [\,]) \\ & = & foldr\ (+)\ 0 & (x_0 : x_1 : x_2 : x_3 : \ldots : x_n : [\,]) \\ & = & & (x_0 + x_1 + x_2 + x_3 + \ldots + x_n + 0) \end{array}$$

Another useful higher-order function is `(.)` — the composition operator we saw earlier, defined by:

```
(f . g) x  =  f (g x)
```

Composition is not of much use when the `x` argument is present; one could achieve the same thing simply by writing the expression in the form `f (g x)`. It is, however, very useful for combining functions without having to refer to their arguments. For example, `square.cube` is a function which when applied to an argument, cubes it and then squares the result. `abs.sum` is a function which calculates the absolute value of the sum of a list of numbers.

⋆ Exercise 22:

What is the effect of applying `foldr (:) []` to a list? Try to work out the answer away from the computer before trying it with GHCi.

⋆ Exercise 23:

The standard prelude function `max` gives the greater of its two arguments. Explain what is wrong with the following definition of `sqmax`, which is meant to square the greater of its arguments:

```
sqmax = square.max
```

□

13 Type Declarations

The type-checker automatically deduces the type of all functions used in a program. Thus, type declarations are completely optional. However, programs are often clearer if they contain some type declarations as well. Also, the type-checker can sometimes give more helpful error messages if it knows the intended types of functions.

Type declarations are written in the form *name*$_1$ `,` ... `,` *name*$_n$ `::` *type*, where the names are those of functions or other values. Some basic types are:

`Int`	integer
`Bool`	boolean
`(`t_1`,`t_2`)`	type t_1 paired with type t_2
`[`t`]`	list of elements of type t
t_1 `->` t_2	function from type t_1 to type t_2

For example:

```
square, cube  ::  Int -> Int
squares       ::  [Int] -> [Int]
(+), (*)      ::  Int -> Int -> Int
```

The arrow representing functions is right-associative, so the type of `(+)` is `Int->(Int->Int)`; when it is applied to an integer (its left-hand operand), the result is a function that can be applied to a second integer (its right-hand operand) yielding an integer result.

Functions that are less restrictive of their argument and result types can use *type variables*:

```
map    ::  (a -> b) -> [a] -> [b]
foldr  ::  (a -> b -> b) -> b -> [a] -> b
```

Note:

- A type variable must consist of a single lower-case character.

If a type contains type variables, it is said to be *polymorphic*. For example, the function `map` has the type given above for any values of `a` and `b`. If we let `a` and `b` both be `Int`, then we see that one possible type of `map` is `(Int->Int) -> [Int] -> [Int]`, so that the application `map square` is well-typed, and itself has the type `[Int]->[Int]`.

⋆ **Exercise 24:**
Add type declarations to your definitions (functions and constant values) from previous exercises. After adding each type declaration, reload the script file to check whether the GHCi type-checker complains.

⋆ **Exercise 25:**
(Challenge!) Using only pencil and paper, work out the type of the composition operator (.). Remember that each of its function arguments may have type `a -> b` rather than `a -> a`. Confirm your answer by executing `:type (.)` in a GHCi session.

□

14 Type Abbreviations

One may give a new name to an existing type by a declaration of the form `type` *name* `=` *type*. The name may subsequently be used anywhere in place of the equivalent type. For example,

```
type  MyString = [Char]

addstop   ::  MyString -> MyString
addstop s  =  s ++ "."
```

Note:

- Type names **must** begin with a capital letter.

□

15 User-Defined Types

The user may define new types. Here is a definition of a tree data type:

```
data  Tree a  =  Leaf a  |  Pair (Tree a) (Tree a)
```

It might be read as follows: "A *Tree* of *a* is either a *Leaf*, which contains an *a*, or a *Pair*, which contains a Tree of *a* and a Tree of *a*." Here *a* is what is called a *generic type variable*; it stands for the type of elements of the tree, which may be any type. *Pair* and *Leaf* are called *constructors*.

Note:

- A constructor name **must** begin with a capital letter.

The command `:type` can be used to find the type of any expression. For example:

```
*Main> :type Pair (Leaf 'a') (Leaf 'b')
Pair (Leaf 'a') (Leaf 'b') :: Tree Char
*Main> :type Leaf (Pair (Leaf "stop") (Leaf "go"))
Leaf (Pair (Leaf "stop") (Leaf "go")) :: Tree (Tree [Char])
```

And recall that in Haskell, a function is any expression that can be applied to another expression. So `Leaf` and `Pair` are considered to be functions:

```
*Main> :type Leaf
Leaf ::  a -> Tree a
*Main> :type (map Leaf)
(map Leaf) ::  [a] -> [Tree a]
```

But `Pair (Leaf "stop") (Leaf 'b')` is not a legal tree (the declaration of `Tree` demands that all leaves in a tree must have elements of the same type), and will cause a type error to be reported:

```
*Main> :type Pair (Leaf "stop") (Leaf 'b')

<interactive>:1:26:
    Couldn’t match expected type ‘[Char]’ with actual type ‘Char’
    In the first argument of ‘Leaf’, namely ‘'b'’
    In the second argument of ‘Pair’, namely ‘(Leaf 'b')’
    In the expression: Pair (Leaf "stop") (Leaf 'b')
```

The constructors `Leaf` and `Pair` may appear on the left-hand side of equations, as in the following function definition:

```
count (Leaf x)    =  1
count (Pair x y)  =  count y + count x
```

For example, `count (Pair (Leaf 'o') (Leaf 'h'))` yields the value 2.

New types do not necessarily involve type variables. For example, the type `Bool` is defined (in the standard prelude) as

```
data  Bool = False | True
```

⋆ Exercise 26:

Write a function that gives a list of the values at the leaves of a Tree. **Hint:** There are two forms of tree to consider:

```
leaves              ::  Tree a -> [a]
leaves (Leaf x)      =  ...
leaves (Pair t0 t1)  =  ...
```

Remember that the right-hand side of each equation must have a list type.

⋆ Exercise 27:

Write a function `treemap` which applies a function to each leaf value in a tree, just as `map` applies a function to each element in a list. **Hint:** The result will be a tree with the same structure as the argument (but different leaf values):

```
treemap                  ::  (a -> b) -> Tree a -> Tree b
treemap f (Leaf x)        =  Leaf (...)
treemap f (Pair t0 t1)    =  Pair (...)  (...)
```

To test the above, define a Tree-valued constant in your script file; for example:

```
t0 ::  Tree Int
t0  = Pair (Pair (Pair (Leaf 3) (Pair (Leaf 0) (Leaf 1)))
                 (Pair (Leaf 5) (Leaf 8)))
           (Pair (Leaf 1) (Leaf 1))
```

You could use a function such as

```
showTree (Leaf x)   = show x
showTree (Pair l r) = "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

to show the results of your calculation.

Now in the session, try evaluating such expressions as `leaves t0`, `treemap square t0`, and `leaves (treemap square t0)`.

□

16 Comments

There are two ways of indicating comments in a Haskell script.

1. Anything between the symbol `--` and the end of the line is treated as a comment.
2. Any text between the symbol `{-` and the symbol `-}` is treated as a comment.

Note that either form of comment can contain further comments within it – the whole outermost comment will be ignored by the interpreter.

□

17 Errors

GHCi always reduces an expression as far as possible. If an expression cannot be further reduced because it is in error, the interpreter will mark it as such, and stop evaluating.

Here is a sample from a session:

```
*Main> [2 `div` x | x<-[-4..4]]
[-1,-1,-1,-2,*** Exception: divide by zero
```

Lazy evaluation allows GHCi to compute the first four list elements before reporting an error.

Haskell has a built-in function called `error` whose type is `String->a`. This is a somewhat odd function: From its type it looks as if it is producing a value of a polymorphic type about which it knows nothing, since it never receives a value of that type as an argument!

In fact, there is one value "shared" by all types: $\perp$ (bottom). Indeed, semantically that is exactly what value is always produced by `error` (all errors have the value $\perp$). However, we can expect that a reasonable implementation will print the string argument to error for diagnostic purposes. Thus this function is useful when we wish to terminate a program when something has "gone wrong." For example, a sensible way to define the `head` function on lists would be:

```
head (x:xs)  =  x
head []      =  error "head []"
```

□

Appendix: Some Useful Session Commands

Commands may be abbreviated to the part shown in **boldface**).

:load *filenames*	load modules from specified files
:load	clear all files except prelude
:add *filenames*	read additional script files
:reload	repeat last load command
:edit *filename*	edit file
:edit	edit last module
expr	evaluate expression
:type *expr*	print type of expression
:?	display full list of commands
:browse	list all names currently in scope
:cd *dir*	change directory
:set +t	display type of every evaluated expression
:unset +t	don't display type of every evaluated expression
:set +s	display time & memory used in expression evaluations
:unset +s	don't display time & memory used in expression evaluations
:quit	exit GHCi interpreter