Chapter 2

Expressions, types and values

In Haskell every *well-formed* expression has, by definition, a well-formed *type*. Each well-formed expression has, by definition, a *value*. Given an expression for evaluation,

- GHCi checks that the expression is *syntactically* correct, that is, it conforms to the rules of syntax laid down by Haskell.
- If it is, GHCi infers a type for the expression, or checks that the type supplied by the programmer is correct.
- Provided the expression is well-typed, GHCi evaluates the expression by reducing it to its simplest possible form to produce a value. Provided the value is printable, GHCi then prints it at the terminal.

In this chapter we continue the study of Haskell by taking a closer look at these processes.

2.1 A session with GHCi

One way of finding out whether or not an expression is well-formed is of course to use GHCi. There is a command `:type expr` which, provided `expr` is well-formed, will return its type. Here is a session with GHCi (with some of GHCi's responses abbreviated):

```
ghci> 3 +4)
<interactive>:1:5: parse error on input `)'
```

GHCi is complaining that on line 1 the character `')'` at position 5 is unexpected; in other words, the expression is not syntactically correct.

```
ghci> :type 3+4
3+4 :: Num a => a
```

GHCi is asserting that the type of 3+4 is a number. More on this below.

```
ghci> :type if 1==0 then 'a' else "a"
<interactive>:1:23:
Couldn't match expected type `Char' with actual type `[Char]'
In the expression: "a"
In the expression: if 1 == 0 then 'a' else "a"
```

GHCi expects the types of `expr1` and `expr2` in a conditional expression

```
    if test then expr1 else expr2
```

to be the same. But a character is not a list of characters so the conditional expression, though conforming to the rules of Haskell syntax, is not well-formed.

```
ghci> sin sin 0.5
<interactive>:1:1:
No instance for (Floating (a0 -> a0))
arising from a use of `sin'
Possible fix: add an instance declaration for
  (Floating (a0 -> a0))
In the expression: sin sin 0.5
In an equation for `it': it = sin sin 0.5
```

GHCi gives a rather opaque error message, complaining that the expression is not well-formed.

```
ghci> sin (sin 0.5)
0.4612695550331807
```

Ah, GHCi is happy with this one.

```
ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

GHCi returns the type of the function `map`.

```
ghci> map
<interactive>:1:1:
No instance for (Show ((a0 -> b0) -> [a0] -> [b0]))
arising from a use of `print'
Possible fix:
add an instance declaration for
```

```
  (Show ((a0 -> b0) -> [a0] -> [b0]))
In a stmt of an interactive GHCi command: print it
```

GHCi is saying that it doesn't know how to print a function.

```
ghci> :type 1 `div` 0
1 `div` 0 :: Integral a => a
```

GHCi is asserting that the type of `1 `div` 0` is an integral number. The expression `1 `div` 0` is therefore well-formed and possesses a value.

```
ghci> 1 `div` 0
*** Exception: divide by zero
```

GHCi returns an error message. So what is the value of `1 `div` 0`? The answer is that it is a special value, written mathematically as $\perp$ and pronounced 'bottom'. In fact, Haskell provides a predeclared name for this value, except that it is called `undefined`, not bottom.

```
ghci> :type undefined
undefined :: a
ghci> undefined
*** Exception: Prelude.undefined
```

Haskell is not expected to produce the value $\perp$. It may return with an error message, or remain perpetually silent, computing an infinite loop, until we interrupt the computation. It may even cause GHCi to crash. Oh, yes.

```
ghci> x*x where x = 3
<interactive>:1:5: parse error on input `where'

ghci> let x = 3 in x*x
9
```

A `where` clause does *not* qualify an expression in Haskell, but the whole of the right-hand side of a definition. Thus the first example is not a well-formed expression. On the other hand, a `let` expression

```
   let <defs> in <expr>
```

is well-formed, at least assuming the definitions in `<defs>` are and the expression `<expr>` is. Let-expressions appear infrequently in what follows, but occasionally they can be useful.

2.2 Names and operators

As we have seen, a script is a collection of names and their definitions. Names for functions and values begin with a lowercase letter, except for data constructors (see later on) which begin with an uppercase letter. Types (e.g. `Int`), type classes (e.g. `Num`) and modules (e.g. `Prelude` or `Data.Char`) also begin with an uppercase letter.

An operator is a special kind of function name that appears between its (two) arguments, such as the + in `x + y` or the ++ in `xs ++ ys`. Operator names begin with a symbol. Any (non-symbolic) function of two arguments can be converted into an operator by enclosing it in back quotes, and any operator can be converted to a prefix name by enclosing it in parentheses. For example,

```
3 + 4     is the same as   (+) 3 4
div 3 4   is the same as   3 `div` 4
```

Operators have different levels of precedence (binding power). For example,

```
3 * 4 + 2       means   (3 * 4) + 2
xs ++ yss !! 3  means   xs ++ (yss !! 3)
```

If in any doubt, add parentheses to remove possible ambiguity. By the way, we can use any names we like for lists, including `x`, `y`, `goodylist`, and so on. But a simple aid to memory is to use `x` for things, `xs` for lists of things, and `xss` for lists of lists of things. That explains why we wrote `yss` in the expression `yss !! 3` in the last line above.

Operators with the same level of precedence normally have an order of association, either to the left or right. For example, the usual arithmetic operators associate to the left:

```
3 - 4 - 2  means  (3 - 4) - 2
3 - 4 + 2  means  (3 - 4) + 2
3 / 4 * 5  means  (3 / 4) * 5
```

Functional application, which has higher precedence than any other operator, also associates to the left:

```
eee bah gum    means  (eee bah) gum
eee bah gum*2  means  ((eee bah) gum)*2
```

Some operators associate to the right:

```
(a -> b) -> [a] -> [b]   means   (a -> b) -> ([a] -> [b])
x ^ y ^ z                means   x ^ (y ^ z)
eee . bah . gum          means    eee . (bah . gum)
```

Of course, if an operator, such as functional composition, is associative the order has no effect on meaning (i.e. the value is the same). Again, one can always add parentheses to remove possible ambiguity.

We can declare new operators; for example:

```
(+++) :: Int -> Int -> Int
x +++ y = if even x then y else x + y
```

The conditional expression has low binding power, so the expression above means

```
if even x then y else (x + y)
```

not `(if even x then y else x) + y`. Again, one can always use parentheses to group differently.

If we like we can declare a precedence level and an order of association for `(+++)`, but we won't spell out how.

Sections and lambda expressions

It is a matter of style, but in the main we prefer to write scripts in which all the little helper functions are named explicitly. Thus if we need a function that adds 1 to a number, or doubles a number, then we might choose to name such functions explicitly:

```
succ, double :: Integer -> Integer
succ n   = n+1
double n = 2*n
```

However, Haskell provides alternative ways of naming these two functions, namely `(+1)` and `(2*)`. The device is called a *section*. In a section one of the arguments of an operator is included along with the operator. Thus

```
(+1) n = n+1
(0<) n = 0<n
(<0) n = n<0
(1/) x = 1/x
```

Sections are certainly attractive ways of naming simple helper functions and we henceforth accept them onto our list of Good Things to Use in Moderation.

There is one important caveat about sections: although `(+1)` is the section that adds 1 to a number, `(-1)` is *not* the section that subtracts 1. Instead `(-1)` is just the number -1. Haskell uses the minus sign both as the binary operation of subtraction and as a prefix to denote negative numbers.

Now suppose we want a function that doubles a number and then adds 1 to the answer. This function is captured by the composition `(+1) . (*2)` of two sections. But the result is unsatisfying because it looks a little abstruse; anyone reading it would have to pause for a moment to see what it meant. The alternative seems to be to give the function a name, but what would be a suitable name? Nothing helpful really comes to mind.

The alternative is to use a *lambda expression* `\n -> 2*n+1`. It is called a lambda expression because mathematically the function would be written as $\lambda n.2{*}n{+}1$. Read the expression as 'that function of n which returns $2{*}n{+}1$'. For example,

```
ghci>  map (\n -> 2*n+1) [1..5]
[3,5,7,9,11]
```

Once in a while a lambda expression seems the best way to describe some function, but only once in a while and we will take them out of the box only on rare occasions.

2.3 Evaluation

Haskell evaluates an expression by reducing it to its simplest possible form and printing the result. For example, suppose we have defined

```
    sqr :: Integer -> Integer
    sqr x = x*x
```

There are basically two ways to reduce the expression `sqr (3+4)` to its simplest possible form, namely 49. Either we can evaluate 3+4 first, or else apply the definition of `sqr` first:

```
   sqr (3+4)                        sqr (3+4)
 = sqr 7                          = let x = 3+4 in x*x
 = let x = 7 in x*x               = let x = 7 in x*x
 = 7*7                            = 7*7
 = 49                             = 49
```

The number of reduction steps is the same in each case, but the order of the reduction steps is slightly different. The method on the left is called *innermost reduction* and also *eager evaluation*; the one on the right is called *outermost reduction* or *lazy evaluation*. With eager evaluation arguments are always evaluated before a function is applied. With lazy evaluation the definition of a function is installed at once and only when they are needed are the arguments to the function evaluated.

Doesn't seem much of a difference, does it? But consider the following (slightly abbreviated) evaluation sequences concerning the function `fst` that returns the first element of a pair, so `fst (x,y) = x`:

```
  fst (sqr 1,sqr 2)        fst (sqr 1,sqr 2)
= fst (1*1,sqr 2)        = let p = (sqr 1,sqr 2)
= fst (1,sqr 2)            in fst p
= fst (1,2*2)            = sqr 1
= fst (1,4)              = 1*1
= 1                      = 1
```

The point here is that under eager evaluation the value `sqr 2` is computed, while under lazy evaluation that value is not needed and is not computed.

Now suppose we add the definitions

```
infinity :: Integer
infinity = 1 + infinity

three :: Integer -> Integer
three x = 3
```

Evaluating `infinity` will cause GHCi to go into a long, silent think trying to compute `1 + (1 + (1 + (1 + (1 + ....` until eventually it runs out of space and returns an error message. The value of `infinity` is ⊥.

Again there are two ways to evaluate `three infinity`:

```
  three infinity                 three infinity
= three (1+infinity)           = let x = infinity in 3
= three (1+(1+infinity))       = 3
= ...
```

Here eager evaluation gets stuck in a loop trying to evaluate `infinity`, while lazy evaluation returns the answer 3 at once. We don't need to evaluate the argument of `three` in order to return 3.

One more definition, a version of the factorial function:

```
factorial :: Integer -> Integer
factorial n = fact (n,1)

fact :: (Integer,Integer) -> Integer
fact (x,y) = if x==0 then y else fact (x-1,x*y)
```

This is another example of a *recursive definition* (the definition of `infinity` was also recursive, and so was the function `song` in the previous chapter). Expressions involving recursive functions are evaluated like any other definition.

Here the two evaluation schemes result in the following sequence of reduction steps (we hide the steps involving simplification of the conditional expression to make another point):

```
  factorial 3                 factorial 3
= fact (3,1)                = fact (3,1)
= fact (3-1,3*1)            = fact (3-1,3*1)
= fact (2,3)                = fact (2-1,2*(3*1))
= fact (2-1,2*3)            = fact (1-1,1*(2*(3*1)))
= fact (1,6)                = 1*(2*(3*1))
= fact (1-1,1*6)            = 1*(2*3)
= fact (0,6)                = 1*6
= 6                         = 6
```

The point to appreciate is that, while the number of reduction steps is basically the same, lazy evaluation requires much more space to achieve the answer. The expression `1*(2*(3*1))` is built up in memory before being evaluated.

The pros and cons of lazy evaluation are briefly as follows. On the plus side, lazy evaluation terminates whenever *any* reduction order terminates; it never takes more steps than eager evaluation, and sometimes infinitely fewer. On the minus side, it can require a lot more space and it is more difficult to understand the precise order in which things happen.

Haskell uses lazy evaluation. ML (another popular functional language) uses eager evaluation. Exercise D explores why lazy evaluation is a Good Thing. Lazy evaluation is considered further in Chapter 7.

A Haskell function `f` is said to be *strict* if `f undefined = undefined`, and *non-strict* otherwise. The function `three` is non-strict, while `(+)` is strict in both arguments. Because Haskell uses lazy evaluation we can define non-strict functions. That is why Haskell is referred to as a *non-strict* functional language.

2.4 Types and type classes

Haskell has built-in (or primitive) types such as `Int`, `Float` and `Char`. The type `Bool` of boolean values is defined in the standard prelude:

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

This is an example of a *data declaration*. The type `Bool` is declared to have two data *constructors*, `False` and `True`. The type `Bool` has three values, not two: `False`, `True` and `undefined :: Bool`. Why do we need that last value? Well, consider the function

```
to :: Bool -> Bool
to b = not (to b)
```

The prelude definition of `not` is

```
not :: Bool -> Bool
not True  = False
not False = True
```

The definition of `to` is perfectly well-formed, but evaluating `to True` causes GHCi to go into an infinite loop, so its value is $\perp$ of type `Bool`. We will have much more to say about data declarations in future chapters.

Haskell has built-in compound types, such as

`[Int]`	a list of elements, all of type `Int`
`(Int,Char)`	a pair consisting of an `Int` and a `Char`
`(Int,Char,Bool)`	a triple
`()`	an empty tuple
`Int -> Int`	a function from `Int` to `Int`

The sole inhabitant of the type `()` is also denoted by `()`. Actually, there is a second member of `()`, namely `undefined :: ()`. Now we can appreciate that there is a value $\perp$ for every type.

As we have already said, when defining values or functions it is always a good idea to include the type signature as part of the definition.

Consider next the function `take n` that takes the first `n` elements of a list. This function made its appearance in the previous chapter. For example,

```
take 3 [1,2,3,4,5] = [1,2,3]
take 3 "category"  = "cat"
take 3 [sin,cos]   = [sin,cos]
```

What type should we assign to `take`? It doesn't matter what the type of the elements of the list is, so `take` is what is called a *polymorphic* function and we denote its type by

```
take :: Int -> [a] -> [a]
```

The `a` is a *type variable*. Type variables begin with a lowercase letter. Type variables can be instantiated to any type.

Similarly,

```
(++) :: [a] -> [a] -> [a]
map  :: (a -> b) -> [a] -> [b]
(.)  :: (b -> c) -> (a -> b) -> (a -> c)
```

The last line declares the polymorphic type of functional composition.

Next, what is the type of `(+)`? Here are some suggestions:

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
(+) :: a -> a -> a
```

The first two types seem too specific, while the last seems too general: we can't add two functions or two characters or two booleans, at least not in any obvious way.

The answer is to introduce *type classes*:

```
(+) :: Num a => a -> a -> a
```

This declaration asserts that `(+)` is of type `a -> a -> a` for any *number type* `a`. A type class, such as `Num`, has a collection of named methods, such as `(+)`, which can be defined differently for each instance of the type class. Type classes therefore provide for *overloaded* functions, functions with the same name but different definitions. Overloading is another kind of polymorphism.

Numbers are rather complicated, and are explained in more detail in the following chapter, so we illustrate type classes with a simpler type class

```
class Eq a  where
  (==),(/=) :: a -> a -> Bool
  x /= y    = not (x == y)
```

This introduces the Equality type class, members of which can use one and the same equality test `(==)` and inequality test `(/=)`. There is a *default* definition of `(/=)` as part of the class, so we only have to provide a definition of `(==)`.

To become a member of the Eq club we have to define an *instance*. For example,

```
instance Eq Bool  where
  x == y  = if x then y else not y

instance Eq Person  where
  x == y  = (pin x == pin y)
```

If pin :: Person -> Pin then we need Eq Pin for the last instance to be correct. Of course, we don't have to make Person a member of the Equality club; we can always define

```
samePerson :: Person -> Person -> Bool
samePerson x y = (pin x == pin y)
```

But we can't use (==) instead of samePerson unless we make an instance declaration.

Here are simplified versions of two other type classes, Ord and Show:

```
class (Eq a) => Ord a  where
  (<),(<=),(>=),(>) :: a -> a -> Bool
  x < y  = not (x >= y)
  x <= y = x == y || x < y
  x >= y = x == y || x > y
  x > y  = not (x <= y)

class Show a  where
  show :: a -> String
```

The boolean operator (||) denotes disjunction: a || b is true only if at least one of a and b is true. We can define this operator by

```
(||) :: Bool -> Bool -> Bool
a || b = if a then True else b
```

The default definitions of the Ord methods are mutually dependent, so one has to provide a specific definition of at least one of them in any instance to break the dependency (unlike Eq where only (/=) was given a default definition). The type class Ord needs Eq as a *superclass* because it makes use of (==) in the default definitions of the four comparison operations.

The type class Show is used for displaying results. Haskell cannot display the result of a computation unless the type of the result is a member of Show. Let us explain this in a little more detail.

2.5 Printing values

We begin with a mystery:

```
ghci> "Hello ++"\n"++ "young" ++"\n"++ "lovers"
"Hello\nyoung\nlovers"
```

Oh. What we wanted was

```
Hello
young
lovers
```

Why didn't Haskell print that?

The reason is that after evaluating a well-formed expression to produce a value, Haskell applies `show` to the value to produce a string that can be printed at the terminal. Applying `show` to a value `v` produces a string that when printed looks exactly like `v`: Thus,

```
   show 42          = "42"
   show 42.3        = "42.3"
   show 'a'         = "'a'"
   show "hello\n"   = "\"hello\\n\""
```

Printing the result involves the use of a Haskell *command*

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

The type `IO a` is a special type, the type of input–output computations that when executed have some interaction with the outside world and return a value of type `a`. If the return value is uninteresting, as with `putStrLn`, we use the null-tuple value `()`.

So, Haskell uniformly applies a show-and-put strategy to print values. Since the greeting above is already a string, we really want to miss out the show step and go straight to the put:

```
ghci> putStrLn ("Hello ++"\n"++ "young" ++"\n"++ "lovers")
Hello
young
lovers
```

Haskell provides many more commands for input–output, for reading and writing to files, for displaying graphics, and so on. Such commands have to be sequenced correctly, and for this Haskell provides a special notation, called `do`-notation. Commands are the subject of Chapter 10, and what follows is simply a foretaste of things to come.

To see an example, consider the common words problem of the previous chapter. There we defined a function

```
commonWords :: Int -> String -> String
```

such that `commonWords n` took a text string and returned a string giving a table of the `n` most common words in the text. The following program reads the text from a file, and writes the output to a file. The type `FilePath` is another synonym for a list of characters:

```
cwords :: Int -> FilePath -> FilePath -> IO()
cwords n infile outfile
   = do {text <- readFile infile;
         writeFile outfile (commonWords n text);
         putStrLn "cwords done!"}
```

Evaluating, for example

```
ghci> cwords 100 "c:\\WarAndPeace" "c:\\Results"
```

on a Windows platform will cause the file `c:\WarAndPeace` to be read, and the results printed to `c:\Results`. The program also prints a message to the terminal. The two component functions of the definition above have types

```
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Suppose that we didn't want to call `cwords` from within an interactive session, but to use it as a stand-alone program. Here is one way. We need to define a value for an identifier `main` of type `IO ()`. Here is such a program:

```
main
  = do {putStrLn "Take text from where:";
        infile <- getLine;
        putStrLn "How many words:";
        n <- getLine;
        putStrLn "Put results where:";
        outfile <- getLine;
        text <- readFile infile;
        writeFile outfile (commonWords (read n) text);
        putStrLn "cwords done!" }
```

For an explanation of `read` see Exercise H. Suppose the common words script is stored in the file `cwords.lhs`. We can compile it with GHC, the Glasgow Haskell Compiler:

```
$ ghc cwords.lhs
```

The compiled program will be stored in the file `cwords.exe`. To run the program under Windows, type

```
$ cwords
```

and follow the instructions.

2.6 Modules

Suppose we thought that the function `commonWords` was sufficiently useful that we wanted to incorporate it into other scripts. The way to do this is to turn the common words script into a *module*. First, we rewrite the script in the following way:

```
module CommonWords (commonWords) where
import Data.Char (toLower)
import Data.List (sort,words)
...
commonWords :: Int -> String -> String
...
```

The `module` declaration is followed by the name of the module, which must begin with a capital letter. Furthermore, the script has to be stored in a file called `CommonWords.lhs` to enable Haskell to find the module (at least, if you are using literate scripts; otherwise it would be `CommonWords.hs`). Following the name of the module is a list of *exports*, the functions, types and other values you want to be able to export to other scripts. The list of exports has to be enclosed in parentheses. Here we just export one function, `commonWords`. The exports are the only things defined in the module that are visible in other modules. Omitting the export list, and the surrounding parentheses, means that everything in the module is exported.

We can then compile the module using GHC and then import it into other scripts with the declaration

```
import CommonWords (commonWords)
```

There are two major advantages of Haskell modules. One is we can structure our scripts into bite-sized chunks, separating out little groups of related functions into separate modules. The other advantage is that the functions in a compiled module are much faster to evaluate because their definitions are compiled into machine-specific code, leading to a much slicker reduction process. GHCi is an *interpreter* rather than a compiler; it evaluates internal forms of expression that are much closer to the source language of Haskell.

2.7 Haskell layout

The examples of do-notation used braces ({ and }) and semicolons; these are examples of *explicit layout*. Braces and semicolons are used only to control layout and have no meaning as part of the language of Haskell expressions. We can use them in other places too:

```
roots :: (Float,Float,Float) -> (Float,Float)
roots (a,b,c)
 | a == 0       = error "not quadratic"
 | disc < 0     = error "complex roots"
 | otherwise    = ((-b-r)/e, (-b+r)/e)
 where {disc = b*b - 4*a*c; r = sqrt d; e = 2*a}
```

Here the where clause uses explicit braces and semicolons rather than appealing to Haskell's layout rules. Instead, we could have written

```
where disc = b*b - 4*a*c
      r    = sqrt d
      e    = 2*a
```

But we couldn't have written

```
where disc = b*b - 4*a*c
         r = sqrt d
         e = 2*a
```

The layout (or *offside*) rule takes effect whenever the opening brace is omitted after the keyword where or do (and also after let). When this happens the indentation of the next item, whether or not on a new line, is remembered. For each subsequent line, if it is indented more, then the previous line is continued; if it is indented the same amount, then a new item begins; and if it is indented less, then the layout list is ended. At least, that's roughly the offside rule.

The offside rule explains why there is an indentation in the declarations of type classes and instances:

```
class Foo a  where
   I am part of the class declaration.
   So am I.
Now the class declaration has ended.
```

You can always put in braces and semicolons if in any doubt. Actually the offside rule can still cause confusion when used with do-notation. So the recommendation is belts, braces and semicolons.

And you thought the football offside rule was complicated.

2.8 Exercises

Exercise A

On the subject of precedence, this question comes from Chris Maslanka's puzzle page in the *Guardian* newspaper:

'Is a half of two plus two equal to two or three?'

Exercise B

Some of the following expressions are not syntactically correct, while others are syntactically correct but do not have sensible types. Some are well-formed. Which is which? In the case of a well-formed expression, give a suitable type. Assume `double :: Int -> Int`. I suggest you don't use a computer to check your answers, but if you do, be prepared for some strange error messages.

The expressions are:

```
[0,1)
double -3
double (-3)
double double 0
if 1==0 then 2==1
"++" == "+" ++ "+"
[(+),(-)]
[[],[[]],[[[]]]]
concat ["tea","for",'2']
concat ["tea","for","2"]
```

Exercise C

In the good old days, one could write papers with titles such as

‘The morphology of prex – an essay in meta-algorithmics’

These days, journals seem to want all words capitalised:

‘The Morphology Of Prex – An Essay In Meta-algorithmics’

Write a function `modernise :: String -> String` which ensures that paper titles are capitalised as above. Here are some helpful questions to answer first:

1. The function `toLower :: Char -> Char` converts a letter to lowercase. What do you think is the name of the prelude function that converts a letter to uppercase?
2. The function `words :: String -> [Word]` was used in the previous chapter. What do you think the prelude function

   ```
   unwords :: [Word] -> String
   ```

   does? Hint: which, if either, of the following equations should hold?

   ```
   words . unwords = id
   unwords . words = id
   ```
3. The function `head :: [a] -> a` returns the head of a nonempty list, and `tail :: [a] -> [a]` returns the list that remains when the head is removed. Suppose a list has head `x` and tail `xs`. How would you reconstruct the list?

Exercise D

Beaver is an eager evaluator, while Susan is a lazy one.[1] How many times would Beaver evaluate `f` in computing `head (map f xs)` when `xs` is a list of length n? How many times would Susan? What alternative to `head . map f` would Beaver prefer?

The function `filter p` filters a list, retaining only those elements that satisfy the boolean test `p`. The type of `filter` is

```
filter :: (a -> Bool) -> [a] -> [a]
```

Susan would happily use `head . filter p` for a function that finds the first element of a list satisfying `p`. Why would Beaver not use the same expression?

Instead, Beaver would probably define something like

[1] If you don’t know, google ‘lazy susan’ to discover what a lazy susan is.

```
first :: (a -> Bool) -> [a] -> a
first p xs | null xs   = error "Empty list"
           | p x       = ...
           | otherwise = ...
           where x = head xs
```

The function `null` returns `True` on an empty list, and `False` otherwise. When evaluated, the expression `error message` stops execution and prints the string `message` at the terminal, so its value is $\perp$. Complete the right-hand side of Beaver's definition.

What alternative might Beaver prefer to `head . filter p . map f`?

Exercise E

The type `Maybe` is declared in the standard prelude as follows:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

This declaration uses a `deriving` clause. Haskell can automatically generate instances of some standard type classes for some data declarations. In the present case the deriving clause means that we don't have to go through the tedium of writing

```
instance (Eq a) => Eq (Maybe a)
  Nothing == Nothing = True
  Nothing == Just y  = False
  Just x == Nothing  = False
  Just x == Just y   = (x == y)

instance (Ord a) => Ord (Maybe a)
  Nothing <= Nothing = True
  Nothing <= Just y  = True
  Just x <= Nothing  = False
  Just x <= Just y   = (x <= y)
```

The reason why `Nothing` is declared to be less than `Just y` is simply because the constructor `Nothing` comes before the constructor `Just` in the data declaration for `Maybe`.

The reason why the `Maybe` type is useful is that it provides a systematic way of handling failure. Consider again the function

```
first p = head . filter p
```

of the previous exercise. Both Eager Beaver and Lazy Susan produced versions of this function that stopped execution and returned an error message when `first p` was applied to the empty list. That's not very satisfactory. Much better is to define

```
first :: (a -> Bool) -> [a] -> Maybe a
```

Now failure is handled gracefully by returning `Nothing` if there is no element of the list that satisfies the test.

Give a suitable definition of this version of `first`.

Finally, count the number of functions with type `Maybe a -> Maybe a`.

Exercise F

Here is a function for computing x to the power n, where $n \geq 0$:

```
exp :: Integer -> Integer -> Integer
exp x n | n == 0    = 1
        | n == 1    = x
        | otherwise = x*exp x (n-1)
```

How many multiplications does it take to evaluate `exp x n`?

Dick, a clever programmer, claims he can compute `exp x n` with far fewer multiplications:

```
exp x n | n == 0  = 1
        | n == 1  = x
        | even n  = ...
        | odd n   = ...
```

Fill in the dots and say how many multiplications it takes to evaluate the expression `exp x n` by Dick's method, assuming $2^p \leq n < 2^{p+1}$.

Exercise G

Suppose a date is represented by three integers $(day, month, year)$. Define a function `showDate :: Date -> String` so that, for example,

```
showDate (10,12,2013) = "10th December, 2013"
showDate (21,11,2020) = "21st November, 2020"
```

You need to know that `Int` is a member of the type class `Show`, so that `show n` produces a string that is the decimal representation of the integer `n`.

Exercise H

The credit card company Foxy issues cards with ten-digit card-identification numbers (CINs). The first eight digits are arbitrary but the number formed from the last two digits is a checksum equal to the sum of the first eight digits. For example, "6324513428" is a valid CIN because the sum of the first eight digits is 28.

Construct a function `addSum :: CIN -> CIN` that takes a string consisting of eight digits and returns a string of ten digits that includes the checksum. Thus `CIN` is a type synonym for `String`, though restricted to strings of digits. (Note that Haskell type synonyms cannot enforce type constraints such as this.) You will need to convert between a digit character and the corresponding number. One direction is easy: just use `show`. The other direction is also fairly easy:

```
    getDigit :: Char -> Int
    getDigit c = read [c]
```

The function `read` is a method of the type class `Read` and has type

```
    read :: Read a => String -> a
```

The type class `Read` is dual to `Show` and `read` is dual to `show`. For example,

```
ghci> read "123" :: Int
123
ghci> read "123" :: Float
123.0
```

The function `read` has to be supplied with the type of the result. One can always add *type annotations* to expressions in this way.

Now construct a function `valid :: CIN -> Bool` that checks whether an identification number is valid. The function `take` might prove useful.

Exercise I

By definition a *palindrome* is a string that, ignoring punctuation symbols, blank characters and whether or not a letter is in lowercase or uppercase, reads the same forwards and backwards. Write an interactive program

```
    palindrome :: IO ()
```

which, when run, conducts an interactive session, such as

```
ghci> palindrome
Enter a string:
```

```
Madam, I'm Adam
Yes!

ghci> palindrome
Enter a string:
A Man, a plan, a canal - Suez!
No!

ghci> palindrome
Enter a string:
Doc, note I dissent. A fast never prevents a fatness.
I diet on cod.
Yes!
```

The function `isAlpha :: Char -> Bool` tests whether a character is a letter, and `reverse :: [a] -> [a]` reverses a list. The function `reverse` is provided in the standard prelude and `isAlpha` can be imported from the library `Data.Char`.

2.9 Answers

Answer to Exercise A

The answer to Maslanka's puzzle is 'Yes!' This little puzzle has fooled a number of distinguished computer scientists.

Answer to Exercise B

My GHCi session produced (with explanations added):

```
ghci> :type [0,1)
<interactive>:1:5: parse error on input `)'
```

GHCi knows that `')'` is wrong, though it is not smart enough to suggest `']'`.

```
ghci> :type double -3
<interactive>:1:9:
No instance for (Num (Int -> Int))
arising from the literal `3'
Possible fix: add an instance declaration for
  (Num (Int -> Int))
```

```
In the second argument of `(-)', namely `3'
In the expression: double - 3
```

The explanation of the error message is that numerical subtraction `(-)` has type `Num a => a -> a`. For `double - 3` to be well-formed (yes, it was typed as `double -3` but the spaces are not significant here), `double` has to be a number, so the class instance `Num (Int -> Int)` is required. But there isn't one: you cannot sensibly subtract a number from a function.

```
ghci> double (-3)
-6
ghci> double double 0
<interactive>:1:1:
The function `double' is applied to two arguments,
but its type `Int -> Int' has only one
In the expression: double double 0
In an equation for `it': it = double double 0
```

Most of GHCi's error message is clear.

```
ghci> if 1==0 then 2==1

<interactive>:1:18:
parse error (possibly incorrect indentation)
```

Conditional expressions are incomplete without an 'else' clause.

```
ghci> "++" == "+" ++ "+"
True
```

Both sides are well-formed and denote the same list.

```
ghci> [(+),(-)]
<interactive>:1:1:
No instance for (Show (a0 -> a0 -> a0))
arising from a use of `print'
Possible fix:
add an instance declaration for
  (Show (a0 -> a0 -> a0))
In a stmt of an interactive GHCi command: print it
```

To display the value `[(+),(-)]` we have to be able to show its elements. But no way of showing functions has been provided.

```
ghci> :type [[],[[]],[[[]]]]
```

```
[[],[[]],[[[]]]] :: [[[[a]]]]
```

To explain, let the main list have type `[b]`. The first element is a list, so `b=[c]`. The second element is a list of lists, so `c=[d]`. The third element is a list of lists of lists, so `d=[a]`.

```
ghci> concat ["tea","for",'2']
<interactive>:1:21:
Couldn't match expected type `[Char]'
with actual type `Char'
In the expression: '2'
In the first argument of `concat',
namely `["tea", "for", '2']'
In the expression: concat ["tea", "for", '2']
```

The first two elements of the list have type `[Char]`, but the last has type `Char` and that is not allowed.

```
ghci> concat ["tea","for","2"]
"teafor2"
```

Answer to Exercise C

1. `toUpper`, of course.
2. Concatenates the words, putting a single space between them. We have

   ```
   words . unwords = id
   ```

   but not `unwords . words = id`.
3. `[x] ++ xs`.

```
modernise :: String -> String
modernise = unwords . map capitalise . words

capitalise :: Word -> Word
capitalise xs = [toUpper (head xs)] ++ tail xs
```

We will see another way of writing `capitalise` in Chapter 4.

Answer to Exercise D

Computing `head (map f xs)` takes *n* evaluations of `f` under eager evaluation, but only one under lazy evaluation. Beaver would have to exploit the identity `head . map f = f . head`.

Instead of defining `first p = head . filter p`, Beaver might define

```
first :: (a -> Bool) -> [a] -> a
first p xs | null xs   = error "Empty list"
           | p x       = x
           | otherwise = first p (tail xs)
          where x = head xs
```

Instead of defining `first p f = head . filter p . map f`, Beaver might define

```
first :: (b -> Bool) -> (a -> b) -> [a] -> b
first p f xs | null xs   = error "Empty list"
             | p x       = x
             | otherwise = first p f (tail xs)
             where x = f (head xs)
```

The point is that with eager evaluation most functions have to be defined using explicit recursion, not in terms of useful component functions like `map` and `filter`.

Answer to Exercise E

Lazy Susan would probably write

```
first p xs = if null ys then Nothing
             else Just (head ys)
             where ys = filter p xs
```

As to the number of functions of type `Maybe a -> Maybe a`, there are just six. Applied to `Nothing` the function can only return `Nothing` or `undefined`. Applied to `Just x` the function can only return `Nothing` or `Just x` or `undefined`. The point is that we know absolutely nothing about the underlying type, so no new values can be invented. That makes six possible functions in all.

Answer to Exercise F

It takes `n-1` multiplications to evaluate `exp x n`. Dick's method is to exploit the identities $x^{2m} = (x^2)^m$ and $x^{2m+1} = x(x^2)^m$ to obtain a recursive definition:

```
exp x n | n == 0  = 1
        | n == 1  = x
        | even n  = exp (x*x) m
        | odd n   = x*exp (x*x) (m-1)
        where m = n `div` 2
```

This is an example of a *divide and conquer* algorithm. Dick's program takes p multiplications, where $2^p \leq n < 2^{p+1}$. Thus $p = \lfloor \log n \rfloor$, where $\lfloor x \rfloor$ returns the *floor* of a number, the greatest integer no bigger than the number. We will consider the floor function in more detail in the following chapter.

Answer to Exercise G

```
showDate :: Date -> String
showDate (d,m,y) = show d ++ suffix d ++ " " ++
                   months !! (m-1) ++ ", " ++ show y
```

The function `suffix` computes the right suffix:

```
suffix d = if d==1 || d==21 || d==31 then "st" else
           if d==2 || d==22 then "nd" else
           if d==3 || d==23 then "rd" else
           "th"

months = ["January",.......]
```

If you indulged in clever arithmetic to compute `suffix`, then you should realise that Sometimes a Simple Solution is Best.

Answer to Exercise H

One solution is as follows:

```
addSum :: CIN -> CIN
addSum cin =
  cin ++ show (n `div` 10) ++ show (n `mod` 10)
  where n = sum (map fromDigit cin)

valid :: CIN -> Bool
valid cin = cin == addSum (take 8 cin)

fromDigit :: Char -> Int
fromDigit c = read [c]
```

The function `fromDigit` will return a numerical digit given a digit character.

Answer to Exercise I

Here is one solution:

```
import Data.Char (toLower,isAlpha)

palindrome :: IO()
palindrome
 = do {putStrLn "Enter a string:";
       xs <- getLine;
       if isPalindrome xs then putStrLn "Yes!"
       else putStrLn "No!"}

isPalindrome :: String -> Bool
isPalindrome xs = (ys == reverse ys)
  where ys = map toLower (filter isAlpha xs)
```

2.10 Chapter notes

The chapter has referred a number of times to the Haskell 'standard prelude'. This is a collection of basic types, type classes, functions and other values that are indispensible in many programming tasks. For a complete description of the standard prelude, see Chapter 8 of the Haskell report; alternatively, visit

`www.haskell.org/onlinereport/standard-prelude.html`

See `www.haskell.org` for more information on the implementation of functional languages and of Haskell in particular. An older book, *The Implementation of Functional Programming Languages* (Prentice Hall, 1987) by Simon Peyton Jones, is no longer in print, but an online version can be found at

`research.microsoft.com/~simonpj/papers/slpj-book-1987`

Apart from GHC there are other maintained compilers for Haskell, including UHC, the Utrecht Haskell Compiler. See the home page `cs.uu.nl/wiki/UHC`.

On the eager-versus-lazy evaluation debate, read Bob Harper's blog article *The point of laziness*, which can be found at

`existentialtype.wordpress.com/2011/04/24/`

In the blog Harper enumerates some of the reasons why he prefers a strict language. But also read Lennart Augustsson's reply to the post. Augustsson's main point, emphasised in Exercise D, is that under strict evaluation you are forced for efficiency reasons to define most functions by explicit recursion, and therefore lose the ability to build definitions out of simple standard functions. That undercuts our ability to reason about functions by applying general laws about their component functions.

Bob Harper is one of the authors of *The Definition of Standard ML (Revised)* (MIT Press, 1989). ML is a strict functional language. You can find an introduction to ML at

```
www.cs.cmu.edu/~rwh/smlbook/book.pdf
```

Another increasingly popular language is Agda, which is both a dependently-typed functional language and also a proof assistant; see the Agda home page

```
wiki.portal.chalmers.se/agda/pmwiki.php
```

Chris Maslanka writes a regular column in the Saturday edition of the *Guardian* newspaper.