Answers to Questions

Hints may be found on page 189.

Chapter 1 (Starting Off)

1

The expression `17` is a number and so is a value already. The expression `1 + 2 * 3 + 4` will evaluate to the value `11`, since the multiplication has higher precedence than addition. The expression `400 > 200` evaluates to the boolean `True` since this is result of the comparison operator `>` on the operands `400` and `200`. Similarly, `1 /= 1` evaluates to `False`. The expression `True || False` evaluates to `True` since one of the operands is true. Similarly, `True && False` evaluates to `False` since one of the operands is false. The expression `if True then False else True` evaluates to `False` since the first (**`then`**) part of the conditional expression is chosen, and takes the place of the entire expression. The expression `'%'` is a character and is already a value.

2

We need to put parentheses around the negative number `-1` as described in the chapter text. The expression `A == a` was probably meant to be `'A' == 'a'`. `False` and `True` need capital letters. An **`if`** ... **`then`** ... **`else`** ... construct must have an **`else`** ... part. The expression `'a' + 'b'` gives an error because the `+` operator does not operate on characters.

3

The expression evaluates to `11`. The programmer seems to be under the impression that spacing affects precedence. It does not, and so this use of space is misleading.

4

The `` `rem` `` operator is of higher precedence than the `+` operator. So `1 + 2 `rem` 3` and `1 + (2 `rem` 3)` are the same expression, evaluating to `1 + 2` which is `3`, but `(1 + 2) `rem` 3` is the same as `3 `rem` 3`, which is `0`.

5

It prevents unexpected values: what would happen if a number other than `1` or `0` was calculated in the program – what would it mean? It is better just to use a different sort of value. We can then show more

easily that a program is correct, since the number of possibilities has for the value has been reduced to just two.

6

The lowercase characters are in alphabetical order, for example `'p' < 'q'` evaluates to `True`. The uppercase characters are similarly ordered. The uppercase letters are all "smaller" than the lowercase characters, so for example `'A' < 'a'` evaluates to `True`. For booleans, `False` is considered "less than" `True`.

Chapter 2 (Names and Functions)

1

Just take in a number and return the number multiplied by ten. The function takes and returns a number, so the type is **Num** a $\Rightarrow$ a $\rightarrow$ a.

```
GHCi:
Prelude> timesTen x = x * 10
Prelude> timesTen 7
70
```

2

We must take two arguments, and use the && and /= operators to test if they are both non-zero. So the result will be of type **Bool**. Each of the arguments must be a number, and must be able to be compared for equality with 0, so each must be of a type which is an instance of the typeclass **Num** and the typeclass **Eq**. It is not required that the two arguments are of the same type, though, since they are not operands to the same operator. The whole type will therefore be (**Eq** a, **Eq** b, **Num** a, **Num** b) $\Rightarrow$ a $\rightarrow$ b $\rightarrow$ **Bool**.

```
GHCi:
Prelude> :{
Prelude| bothNonZero x y =
Prelude|   x /= 0 && y /= 0
Prelude| :}
Prelude> bothNonZero 10 (-1)
True
```

3

Our function should take a number and return another one (the sum). The base case is when the number is equal to 1. Then, the sum of all numbers from $1 \ldots 1$ is just 1. If not, we add the argument to the sum of all the numbers from $1 \ldots (n-1)$. Since the function must take a number which can be compared with 1, and return the same sort of number, it will have the type (**Eq** a, **Num** a) $\Rightarrow$ a $\rightarrow$ a.

```
GHCi:
Prelude> :{
Prelude| sum' n =
Prelude|   if n == 1 then 1 else n + sum' (n - 1)
Prelude| :}
```

```
Prelude> sum' 10
55
```

The function is recursive. What happens if the number given is zero or negative?

4

A number to the power of 0 is 1. A number to the power of 1 is itself. Otherwise, the answer is the current n multiplied by n^{x-1}.

```
GHCi:
Prelude> :{
Prelude| power x n =
Prelude|   if n == 0 then 1 else (if n == 1 then x else x * power x (n - 1))
Prelude| :}
Prelude> power 2 5
32
```

Both arguments must be numbers. In addition, the second must be testable for equality. The result will have the same type as the first argument, so the type of `power` will be (**Eq** b, **Num** a, **Num** b) ⇒ a → b → a. Notice that we had to put one **`if ... then ... else`** inside the **`else`** part of another to cope with the three different cases. The parentheses are not actually required, though, so we may write it like this:

```
GHCi:
Prelude> :{
Prelude| power x n =
Prelude|   if n == 0 then 1 else if n == 1 then x else x * power x (n - 1)
Prelude| :}
```

We could lay out the **`if`** ... **`then`** ... **`else`** ... construct over three lines like this:

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

Or like this:

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

Which do you find easier to read? We can also remove the case for `n == 1` since `power x 1` will reduce to `x * power x 0` which is just `x`.

5

The function `isConsonant` will have type **Char** → **Bool**. If a lower case character in the range `'a'...'z'` is not a vowel, it must be a consonant. So we can reuse the `isVowel` function we wrote earlier, and negate its result using the `not'` function:

```
GHCi:
Prelude> :{
Prelude| isVowel c =
Prelude|   c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
Prelude|
Prelude| isConsonant c =
Prelude|   not' (isVowel c)
Prelude| :}
Prelude> isConsonant 'x'
True
```

6

The expression is the same as **let** `x = 1` **in** (**let** `x = 2` **in** `x + x`), and so the result is 4. Both instances of `x` in `x + x` evaluate to 2 since this is the value assigned to the name `x` in the nearest enclosing **let** expression.

7

We could simply return 0 for a negative argument. The factorial of 0 is 1, so we can change that too, and say our new function finds the factorial of any non-negative number:

```
Prelude> :{
Prelude| factorial n =
Prelude|   if n < 0 then 0 else
Prelude|   if n == 0 then 1 else
Prelude|   n * factorial (n - 1)
Prelude| :}
Prelude> factorial 0
1
```

The number must be capable of being compared with 0 using equality and ordering. However, being an instance of typeclass **Ord** implies being an instance of typeclass **Eq**, so we need not mention **Eq** and the type is (**Num** a, **Ord** a) ⇒ a → a.

Later in the book, we will learn the proper way to deal with arguments for which there is no sensible answer.

8

`1`	**Num** a ⇒ a
`1 + 2`	**Num** a ⇒ a
`f x y = x < y`	**Ord** a ⇒ a → a → **Bool**
`g x y = x < y + 2`	(**Ord** a, **Num** a) ⇒ a → a → **Bool**
`h x y = 0`	**Num** a ⇒ b → c → a
`i x y z = x + 10`	**Num** a ⇒ a → b → c → a

9

`46 * 10`	::	**Num** a ⇒ a
`2 > 1`	::	**Bool**
`f`	::	**Num** a ⇒ a → a
`g`	::	(**Ord** a, **Num** a) ⇒ a → a → b → **Bool**
`i`	::	a → b → c → b

10

The expression `True + False` is not accepted because addition is not defined on booleans. Or, more formally, because the addition operator requires that its operands have a type which is an instance of the typeclass **Num**. The expression `6 + '6'` is invalid for a similar reason. The character `'6'` and the number `6` are entirely unrelated. The function definition `f x y z = (x < y) < (z + 1)` is invalid because `x < y` has type **Bool** which cannot be compared with the expression `z + 1`.

11

The type **Num** a ⇒ b is not valid, since it tries to constrain a which is not mentioned to the right of the ⇒ symbol. The types **Num** a ⇒ a and **Num** t1 ⇒ t1 are equivalent, since a simple consistent renaming turns one into the other. The types **Num** a ⇒ a → b and **Num** b ⇒ b → a are similarly equivalent, though **Num** a ⇒ a → a is different.

The types (**Num** a, **Ord** a) ⇒ a → a and (**Ord** a, **Num** a) ⇒ a → a are equivalent: the order the typeclass constraints are given in is irrelevant.

12

Constraint **Eq** a is not required. All types in **Ord** are in **Eq**. So the examples read:

Ord a ⇒ a → b → a

(**Ord** a, **Eq** b) ⇒ b → b → a

Chapter 3 (Case by Case)

1

We can just pattern-match on the boolean. It does not matter, in this instance, which order the two cases are in.

```
not' :: Bool → Bool

not' True = False
not' False = True
```

2

Recall our solution from the previous chapter:

```
sum' :: (Eq a, Num a) ⇒ a → a

sum' n =
  if n == 1 then 1 else n + sum' (n - 1)
```

Modifying it to use pattern matching:

```
sumMatch :: (Eq a, Num a) ⇒ a → a

sumMatch 1 = 1
sumMatch n = n + sumMatch (n - 1)
```

Notice that the **Eq** typeclass appears even though we do not use the == operator. This is because the pattern-matching implicitly performs an equality test, to check if the argument is equal to one.

3

Again, modifying our solution from the previous chapter:

```
powerMatch :: (Num a, Num b, Eq b) ⇒ a → b → a

powerMatch _ 0 = 1
powerMatch x 1 = x
powerMatch x n = x * powerMatch x (n - 1)
```

4

We can write `not'` using guarded equations like this.

```
not' :: Bool → Bool

not' x | x == False = True
       | otherwise  = False
```

Or like this:

```
not' :: Bool → Bool

not' x | x          = False
       | otherwise  = True
```

Can you see why? Or, since we do not have to use **otherwise**, like this:

```
not' :: Bool → Bool

not' x | x == False = True
       | x == True  = False
```

The sumMatch rewrite is much the same:

```
sumMatch :: (Eq a, Num a) ⇒ a → a

sumMatch n | n == 1    = 1
           | otherwise = n + sumMatch (n - 1)
```

We can deduce that guarded equations are not really clearer when there are only two cases. The guarded equation version of powerMatch is much better:

```
powerMatch :: (Num a, Num b, Eq b) ⇒ a → b → a

powerMatch x n | n == 0    = 1
               | n == 1    = x
               | otherwise = x * powerMatch x (n - 1)
```

5

We need just three cases, remembering that characters may be compared using the comparison operators:

```
kind :: Num a ⇒ Char → a

kind c | c >= 'a' && c <= 'z' = 0
       | c >= 'A' && c <= 'Z' = 1
       | otherwise            = 2
```

Chapter 4 (Making Lists)

1

This is similar to `oddElements`:

```
evenElements :: [a] → [a]

evenElements []          = []                          list has zero elements
evenElements [_]         = []                          list has one element – drop it
evenElements (_:x:xs)    = x : evenElements xs         keep second element, carry on
```

But we can perform the same trick as before, by reversing the cases, to reduce their number:

```
evenElements :: [a] → [a]

evenElements (_:x:xs)    = x : evenElements xs         drop one, keep one, carry on
evenElements l           = []                          otherwise, no more to drop
```

2

This is like counting the length of a list, but we only count if the current element is `True`.

```
countTrue :: Num a ⇒ [Bool] → a

countTrue []          = 0                              no more
countTrue (True:xs)   = 1 + countTrue xs               count this one
countTrue (False:xs)  = countTrue xs                   but not this one
```

3

To make a palindrome from any list, we can append it to its reverse. To check if a list is a palindrome, we can compare it for equality with its reverse (the comparison operators work over almost all types).

```
makePalindrome :: [a] → [a]
isPalindrome :: Eq a ⇒ [a] → Bool

makePalindrome l =
  l ++ reverse' l

isPalindrome l =
  l == reverse' l
```

4

We pattern match with three cases. The empty list, where we have reached the last element, and where we have yet to reach it.

```
dropLast :: [a] → [a]

dropLast [] = []
dropLast [_] = []                                  it is the last one, so remove it
dropLast (x:xs) = x : dropLast xs                  at least two elements remain
```

5

The empty list cannot contain the element; if there is a non-empty list, either the head is equal to the element we are looking for, or if not, the result of our function is just the same as the result of recursing on the tail.

Note that we are using the property that `||` returns its right hand side if its left hand side is false to limit the recursion – our function really does stop as soon as it finds the element.

```
elem' :: Eq a ⇒ a → [a] → Bool

elem' e [] = False
elem' e (x:xs) = x == e || elem' e xs
```

6

If a list is empty, it is already a set. If not, either the head exists somewhere in the tail or it does not; if it does exist in the tail, we can discard it, since it will be included later. If not, we must include it.

```
makeSet :: Eq a ⇒ [a] → [a]

makeSet [] = []
makeSet (x:xs) = if elem' x xs then makeSet xs else x : makeSet xs
```

For example, consider the evaluation of `makeSet [4, 5, 6, 5, 4]`:

$$
\begin{aligned}
& \texttt{makeSet [4, 5, 6, 5, 4]} \\
\Longrightarrow \quad & \texttt{makeSet [5, 6, 5, 4]} \\
\Longrightarrow \quad & \texttt{makeSet [6, 5, 4]} \\
\Longrightarrow \quad & \texttt{6 : makeSet [5, 4]} \\
\Longrightarrow \quad & \texttt{6 : 5 : makeSet [4]} \\
\Longrightarrow \quad & \texttt{6 : 5 : 4 : makeSet []} \\
\Longrightarrow \quad & \texttt{6 : 5 : 4 : []} \\
\overset{*}{\Longrightarrow} \quad & \texttt{[6, 5, 4]}
\end{aligned}
$$

7

The first part of the evaluation of `reverse'` takes time proportional to the length of the list, processing each element once. However, when the lists are appended together, the order of the operations is such that the first argument becomes longer each time. The `++` operator, as we know, also takes time proportional to the length of its first argument. And so, this accumulating of the lists takes time proportional to the square of the length of the list.

```
        reverse' [1, 2, 3, 4]
⟹      reverse' [2, 3, 4] ++ [1]
⟹      (reverse' [3, 4] ++ [2]) ++ [1]
⟹      ((reverse' [4] ++ [3]) ++ [2]) ++ [1]
⟹      (((reverse' [] ++ [4]) ++ [3]) ++ [2]) ++ [1]
⟹      ((([] ++ [4]) ++ [3]) ++ [2]) ++ [1]
⟹      (([4] ++ [3]) ++ [2]) ++ [1]
⟹      ([4, 3] ++ [2]) ++ [1]
⟹      [4, 3, 2] ++ [1]
⟹      [4, 3, 2, 1]
```

By using an additional argument to collect results – called an *accumulator*, we can write a version which operates in time proportional to the length of the list:

```
revInner :: [a] → [a] → [a]
reverse' :: [a] → [a]

revInner a [] = a
revInner a (x:xs) = revInner (x : a) xs

reverse' l =
  revInner [] l
```

For the same list:

```
        reverse' [1, 2, 3, 4]
⟹      revInner [] [1, 2, 3, 4]
⟹      revInner (1 : []) [2, 3, 4]
⟹      revInner (2 : 1 : []) [3, 4]
⟹      revInner (3 : 2 : 1 : []) [4]
⟹      revInner (4 : 3 : 2 : 1 : []) []
⟹      4 : 3 : 2 : 1 : []
  =     [4, 3, 2, 1]
```

8

Both are `[]`. We can use the `reverse'` function already defined, and write `reverse' [1 .. 10]`. In fact, we may also write `[10,9 .. 1]`.

9

We can use a guard:

```
GHCi:
Prelude> [x | x <- [1 .. 9999], x `rem` 21 == 0 && x `rem` 83 == 0]
[1743,3486,5229,6972,8715]
Prelude> [x | x <- [1 .. 9999], x `rem` 21 == 0 || x `rem` 83 == 0]
[21,42,63,83,84,105,126,147,166,168,189,210,231,249,252,273,294......
```

Instead of the && operator, for the first case, we can use two guards:

```
GHCi:
Prelude> [x | x <- [1 .. 9999], x `rem` 21 == 0, x `rem` 83 == 0]
[1743,3486,5229,6972,8715]
```

10

We can generate the list of just the `True` entries and then count the length, writing:

```
GHCi:
Prelude> countTrue l = length' [x | x <- l, x == True]
Prelude> countTrue [True, False, True]
2
```

But, of course, `x == True` is just the same as `x`, since `x` is a boolean. So we may simplify:

```
GHCi:
Prelude> countTrue l = length' [x | x <- l, x]
Prelude> countTrue [True, False, True]
2
```

Chapter 5 (Sorting Things)

1

We may rewrite with the **`where`** construct. Notice that the name `n` is defined in the first line of the **`where`** construct and used in the second and third lines.

```
mergeSort :: Ord a ⇒ [a] → [a]

mergeSort [] = []                                  we are done if the list is empty
mergeSort [x] = [x]                          and also if it only has one element
mergeSort l =
  merge (mergeSort left) (mergeSort right)          sort and merge them
    where
      n = length' l `div` 2
      left = take' n l                                  get the left hand half
      right = drop' n l                               and the right hand half
```

Alternatively, using **let** to define a name representing the number we will take or drop:

```
mergeSort :: Ord a ⇒ [a] → [a]

mergeSort [] = []                                  we are done if the list is empty
mergeSort [x] = [x]                                and also if it only has one element
mergeSort l =
  let n = length' l `div` 2 in
    let left = take' n l                           get the left hand half
        right = drop' n l                          and the right hand half
    in
      merge (mergeSort left) (mergeSort right)     sort and merge them
```

2

The argument to `take'` or `drop'` is `length' l `div` 2` which is clearly less than or equal to `length' l` for all possible values of `l`. Thus, `take'` and `drop'` always succeed. In our case, `take'` and `drop'` are only called when `length' l` is more than 1, due to the pattern matching.

3

We may simply replace the <= operator with the >= operator in the `insert` function.

```
insert :: Ord a ⇒ a → [a] → [a]

insert x [] = [x]
insert x (y:ys) =
  if x >= y
    then x : y : ys
    else y : insert x ys
```

The `sort` function is unaltered.

4

We require a function of type [a] → **Bool**. List of length zero and one are, by definition, sorted. If the list is longer, check that its first two elements are in sorted order. If this is true, also check that the rest of the list is sorted, starting with the second element.

```
isSorted :: Ord a ⇒ [a] → Bool

isSorted [] = True
isSorted [x] = True
isSorted (x:x':xs) = x <= x' && isSorted (x' : xs)
```

We can reverse the cases to simplify:

```
isSorted :: Ord a ⇒ [a] → Bool

isSorted (x:x':t) = x <= x' && isSorted (x' : xs)
isSorted _ = True
```

5

Lists are compared starting with their first elements. If the elements differ, they are compared, and that is the result of the comparison. If both have the same first element, the second elements are considered, and so on. If the end of one list is reached before the other, the shorter list is considered smaller. For example:

```
[1] < [1,2] < [2] < [2, 1] < [2, 2]
```

These are the same principles you use to look up a word in a dictionary: compare the first letters – if same, compare the second etc. So, when applied to the example in the question, it has the effect of sorting the words into alphabetical order.

6

First, using **`where`**:

```
sortComplete :: Ord a ⇒ [a] → [a]

sortComplete [] = []
sortComplete (x:xs) = insert x (sortComplete xs)
  where
    insert a [] = [a]
    insert a (x:xs) =
      if a <= x then a : x : xs else x : insert a xs
```

Now using the **`let`** … **`in`** … construct:

```
sortComplete :: Ord a ⇒ [a] → [a]

sortComplete [] = []
sortComplete (x:xs) =
  let insert a [] = [a]
      insert a (x:xs) =
        if a <= x then a : x : xs else x : insert a xs
  in
    insert x (sortComplete xs)
```

Chapter 6 (Functions upon Functions upon Functions)

1

Our function will have type **String** → **String**. We just match on the argument character list: if it is empty, we are done. If it starts with an exclamation mark, we output a period, and carry on. If not, we output the character unchanged, and carry on:

```
calm :: String → String

calm [] = []
calm ('!':xs) = '.' : calm xs
calm (x:xs) = x : calm xs
```

To use `map'` instead, we write a simple function `calmChar` to process a single character. We can then use `map'` to build our main function:

```
calmChar :: Char → Char
calm :: String → String

calmChar '!' = '.'
calmChar x = x

calm l =
  map' calmChar l
```

This avoids the explicit recursion of the original, and so it is easier to see what is going on.

2

The `clip` function is of type (**Num** a, **Ord** a) ⇒ a → a and is easy to write:

```
clip :: (Num a, Ord a) ⇒ a → a

clip x =
  if x < 1 then 1 else
    if x > 10 then 10 else x
```

Now we can use `map'` for the `clipList` function:

```
clipList :: (Num a, Ord a) ⇒ [a] → [a]

clipList l =
  map' clip l
```

We can write the `clip` function more neatly with guarded equations:

```
clip :: (Num a, Ord a) ⇒ a → a

clip x | x < 1 = 1
       | x > 10 = 10
       | otherwise = x
```

3

Just put the body of the clip function inside an anonymous function:

```
clipList :: (Num a, Ord a) ⇒ [a] → [a]

clipList l =
  map'
    (\x ->
      if x < 1 then 1 else
        if x > 10 then 10 else x)
    l
```

4

We require a function `apply f n x` which applies function `f` a total of `n` times to the initial value `x`. The base case is when `n` is zero.

```
apply :: (Eq b, Num b) ⇒ (a → a) → b → a → a

apply f 0 x = x                                  just x
apply f n x = f (apply f (n - 1) x)              reduce problem size by one
```

Consider the type:

$$(\textbf{Eq}\ b,\ \textbf{Num}\ b) \Rightarrow \overbrace{(a \to a)}^{\text{function f}} \to \overbrace{b}^{\text{n}} \to \overbrace{a}^{\text{x}} \to \overbrace{a}^{\text{result}}$$

The function `f` must take and return the same type, since its result in one iteration is fed back in as its argument in the next. Therefore, the argument `x` and the final result must also have a suitable type, but must be in typeclass **Eq** (for the comparison with `0`) and typeclass **Num** (for the subtraction of `1`). For example, we might have a power function:

```
power :: (Eq b, Num b, Num a) ⇒ a → b → a

power a b =
  apply (\x -> x * a) b 1
```

So `power a b` calculates a^b. The extra constraint **Num** `b` comes from the $*$ operator used inside the function passed to `apply`.

5

We can add an extra argument to the `insert` function, and use that instead of the comparison operator:

```
insert :: (a → a → Bool) → a → [a] → [a]

insert f x [] = [x]                                   add extra argument f
insert f x (y:ys) =
  if f x y
    then x : y : ys
    else y : insert f x ys                    remember to add f here too
```

Now we just need to rewrite the `sort` function.

```
sort :: (a → a → Bool) → [a] → [a]

sort f [] = []
sort f (x:xs) = insert f x (sort f xs)
```

6

We cannot use `map'` here, because the result list will not necessarily be the same length as the argument list. The function will have type (a → **Bool**) → [a] → [a].

```
filter' :: (a → Bool) → [a] → [a]

filter' f [] = []
filter' f (x:xs) =
  if f x
    then x : filter' f xs
    else filter' f xs
```

For example, `filter' (\x -> x `rem` 2 == 0) [1, 2, 4, 5]` evaluates to `[2, 4]`.

7

The function will have type (a → **Bool**) → [a] → **Bool**.

```
all' :: (a → Bool) → [a] → Bool

all' f [] = True
all' f (x:xs) = f x && all' f xs              true for this one, and all the others
```

For example, we can see if all elements of a list are positive: `all' (\x -> x > 0) [1, 2, -1]` evaluates to `False`. Notice that we are relying on the fact that && only evaluates its right hand side when the left hand side is true to limit the recursion.

8

The function will have type `(a → b) → [[a]] → [[b]]`. We use `map'` on each element of the list.

```
mapl :: (a → b) → [[a]] → [[b]]

mapl f [] = []
mapl f (x:xs) = map' f x : mapl f xs
```

We have used explicit recursion to handle the outer list, and `map'` to handle each inner list.

9

Without the composition operator, we may use the filter function from Question 6, and write:

```
GHCi:
Prelude> f l = reverse' (sort (<) (filter' (\x -> x `rem` 15 == 0) l))
Prelude> f [1 .. 100]
[90,75,60,45,30,15]
```

Now, we can use the composition operator instead:

```
GHCi:
Prelude> f = reverse' . sort (<) . filter' (\x -> x `rem` 15 == 0)
Prelude> f [1 .. 100]
[90,75,60,45,30,15]
```

Note that we have to drop the argument `l` for this to work.

Chapter 7 (When Things Go Wrong)

1

The function `s`, used by `smallest` works through the input list one item at a time, keeping track of the smallest number found so far, and updating it if necessary. When the input list is empty, the smallest element (if any) is returned.

```
smallest :: (Num a, Ord a) ⇒ [a] → Maybe a

smallest x = s Nothing x where
  s Nothing [] = Nothing
  s (Just a) [] = Just a
  s Nothing (x:xs) =
    if x > 0
      then s (Just x) xs
      else s Nothing xs
  s (Just a) (x:xs) =
    if x > 0 && x < a
      then s (Just x) xs
      else s (Just a) xs
```

2

We just surround the call to `smallest`.

```
smallest0 :: (Num a, Ord a) ⇒ [a] → a

smallest0 l =
  case smallest l of
    Nothing -> 0
    Just a -> a
```

3

We write a function `s` which, given a test number `x` and a target number `n` squares `x` and tests if it is more than `n`. If it is, the answer is `x - 1`. The test number will be initialized at `1`. The function `sqrtMaybe` returns `Nothing` if the number is negative and otherwise begins the testing process.

```
sqrtMaybe :: (Num a, Ord a) ⇒ a → Maybe a

sqrtMaybe n =
  if n < 0 then Nothing else Just (s 1 n)
    where
      s x n = if x * x > n then x - 1 else s (x + 1) n
```

4

This is a simple variation on `mapMaybe`. We test the result of applying the function to each element, substituting the default value when `Nothing` is returned.

```
mapMaybeDefault :: (a → Maybe b) → b → [a] → [b]

mapMaybeDefault f _ [] = []
mapMaybeDefault f d (x:xs) =
  case f x of
    Just r -> r : mapMaybeDefault f d xs
    Nothing -> d : mapMaybeDefault f d xs
```

5

We use **let** to assign names to the parts of the tuple returned by processing the rest of the list, then apply the function to the first element, building the new result tuple.

```
splitEither :: (a → Either b c) → [a] → ([b], [c])

splitEither f [] = ([], [])
splitEither f (x:xs) =
  let (ls, rs) = splitEither f xs in
    case f x of
      Left l -> (l : ls, rs)
      Right r -> (ls, r : rs)
```

Chapter 8 (Looking Things Up)

1

Since the keys must be unique, the number of different keys is simply the length of the list representing the dictionary – so we can just use the usual `length'` function.

2

The type is the same as for the `add` function, but with a **Maybe** in the return type. However, if we reach the end of the list, we return `Nothing`, since we did not manage to find the entry to replace.

```
replace :: Eq a ⇒ a → b → [(a, b)] → Maybe [(a, b)]

replace k v [] = Nothing                                  could not find it
replace k v ((k', v'):xs) =
  if k == k' then Just ((k, v) : xs) else
    case replace k v t of                                 found it – replace
      Just xs' -> Just ((k', v') : xs')                   already found and replaced
      Nothing -> Nothing                                  not found
```

3

The function takes a list of keys and a list of values and returns a dictionary if it succeeds. So it will have type [a] → [b] → **Maybe** [(a, b)].

```
makeDict :: [a] → [b] → Maybe [(a, b)]

makeDict [] [] = Just []
makeDict _ [] = Nothing
makeDict [] _ = Nothing
makeDict (k:ks) (v:vs) =
  case makeDict ks vs of
    Nothing -> Nothing
    Just xs -> Just ((k, v) : xs)
```

4

This will have the type [(a, b)] → ([a], [b]). For the first time, we need to return a pair, building up both result lists element by element. This is rather awkward, since we will need the tails of both of the eventual results, so we can attach the new heads.

```
makeLists :: [(a, b)] → ([a], [b])

makeLists [] = ([], [])                          build the empty pair
makeLists ((k, v):xs) = (k : ks, v : vs)    there is at least one key-value pair
  where (ks, vs) = makeLists xs
```

We do this by using the **where** construct to do what is effectively pattern matching on one pattern, allowing it to assign the names ks and vs simultaneously.

Here is a sample evaluation (we cannot really show it in the conventional way, so you must work through it whilst looking at the function definition):

```
      makeLists [(1, 2), (3, 4), (5, 6)]
⟹           makeLists [(3, 4), (5, 6)]
⟹                   makeLists [(5, 6)]
⟹                         makeLists []
⟹                             ([], [])
⟹                           ([5], [6])
⟹                     ([3, 5], [4, 6])
⟹               ([1, 3, 5], [2, 4, 6])
```

5

We can use our `elem'` function which determines whether an element is a member of a list, building up a list of the keys we have already seen, and adding to the result list of key-value pairs only those with new keys.

```
dictionaryOfPairsInner :: Eq a ⇒ [a] → [(a, b)] → [(a, b)]
dictionaryOfPairs :: Eq a ⇒ [(a, b)] → [(a, b)]

dictionaryOfPairsInner keysSeen [] = []
dictionaryOfPairsInner keysSeen ((k, v):xs) =
  if elem' k keysSeen
    then dictionaryOfPairsInner keysSeen xs
    else (k, v) : dictionaryOfPairsInner (k : keysSeen) xs

dictionaryOfPairs l =
  dictionaryOfPairsInner [] l
```

How long does this take to run? Consider how long `elem'` takes.

6

We pattern match on the first list – if it is empty, the result is simply the second list. Otherwise, we add the first element of the first list to the union of the rest of its elements and the second list.

```
union :: Eq a ⇒ [(a, b)] → [(a, b)] → [(a, b)]

union [] ys = ys
union ((k, v):xs) ys = add k v (union xs ys)
```

We can verify that the elements of the first dictionary have precedence over the elements of the second dictionary by noting that `add` replaces a value if the key already exists.

Chapter 9 (More with Functions)

1

The function `g a b c` has type $a \to b \to c \to d$ which can also be written $a \to (b \to (c \to d))$. Thus, it takes an argument of type `a` and returns a function of type $b \to (c \to d)$ which, when you give it an argument of type `b` returns a function of type $c \to d$ which, when you give it an argument of type `c` returns something of type `d`. And so, we can apply just one or two arguments to the function `g` (which is called partial application), or apply all three at once. When we write `g a b c = ...` this is just shorthand for `g = \a -> \b -> \c -> ...`

2

The type of `elem'` is **Eq** a ⇒ a → [a] → **Bool**, so if we partially apply the first argument, the type of `elem' e` must be **Eq** a ⇒ [a] → **Bool**. We can use the partially-applied `elem'` function and `map'` to produce a list of boolean values, one for each list in the argument, indicating whether or not that list contains the element. Then, we can use `elem'` again to make sure there are no `False` booleans in the list.

```
elemAll :: Eq a ⇒ a → [[a]] → Bool

elemAll e ls =
  let booleans = map' (elem' e) ls in
    not' (elem' False booleans)
```

We could also write:

```
elemAll :: Eq a ⇒ a → [[a]] → Bool

elemAll e ls =
  not' (elem' False (map' (elem' e) ls))
```

Here it is using the `.` operator:

```
elemAll :: Eq a ⇒ a → [[a]] → Bool

elemAll e =
  not' . (elem' False) . (map' (elem' e))
```

Which do you think is clearer? Why do we check for the absence of `False` rather than the presence of `True`?

3

The function `map'` has type (a → b) → [a] → [b]. The function `mapl` we wrote has type (a → b) → [[a]] → [[b]]. So the function `mapll` will have type (a → b) → [[[a]]] → [[[b]]]. It may be defined thus:

```
mapll :: (a → b) → [[[a]]] → [[[b]]]

mapll f l = map' (map' (map' f)) l
```

But, as discussed, we may remove the `ls` too:

```
mapll :: (a → b) → [[[a]]] → [[[b]]]

mapll f = map' (map' (map' f))
```

In fact, making use of the `.` operator gives the simplest and neatest form:

```
mapll :: (a → b) → [[[a]]] → [[[b]]]

mapll = map' . map' . map'
```

4

We can write a function to truncate a single list using our `take'` function, being careful to deal with the case where there is not enough to take, and then use this and `map'` to build `truncateLists` itself.

```
truncateList :: (Ord a, Num a) ⇒ a → [b] → [b]
truncateLists :: (Ord a, Num a) ⇒ a → [[b]] → [[b]]

truncateList n l =
  if length' l >= n then take' n l else l

truncateLists n ll = map' (truncateList n) ll
```

Here we have used partial application of `truncateList` to build a suitable function for `map'`.

5

First, define a function which takes the given number and a list, returning the first element (or the number if none). We can then build the main function, using partial application to make a suitable function to give to `map'`:

```
firstElt :: a → [a] → a
firstElts :: a → [[a]] → [a]

firstElt n [] = n
firstElt n (x:_) = x

firstElts n l = map' (firstElt n) l
```

6

We can use `map'` and an operator section, and write:

```
GHCi:
Prelude> addNum n ls = map' (n :) ls
Prelude> addNum 1 [[2], [3, 4]]
[[1,2],[1,3,4]]
```

Chapter 10 (New Kinds of Data)

1

We need two constructors – one for squares, which needs just a single number (the length of a side), and one for rectangles which needs two numbers (the width and height, in that order):

```
data Rect a = Square a
            | Rectangle a a deriving Show
```

The name of our new type is Rect. A Rect is either a `Square` or a `Rectangle`. Just like the Colour type in the chapter text, the type variable `a` is used to stand for the type of the data accompanying the constructors. In our scenario, it will always be a number. For example,

```
s :: Num a ⇒ Rect a
r :: Num a ⇒ Rect a

s = Square 7

r = Rectangle 5 2                                    width 5, height 2
```

2

We pattern match on the argument:

```
area :: Num a ⇒ Rect a → a

area (Square s) = s ∗ s
area (Rectangle w h) = w ∗ h
```

3

This will be a function of type Rect `a` → Rect `a`. Squares remain unaltered, but if we have a rectangle with a bigger width than height, we rotate it by ninety degrees.

```
rotate :: Ord a ⇒ Rect a → Rect a

rotate (Rectangle w h) =
  if w > h then Rectangle h w else Rectangle w h
rotate (Square s) = Square s
```

4

We will use `map'` to perform our rotation on any Rects in the argument list which need it. We will then use the sorting function from the previous chapter which takes a custom comparison function so as to just compare the widths.

```
widthOfRect :: Rect a → a
rectCompare :: Ord a ⇒ Rect a → Rect → Bool
pack :: Ord a ⇒ [Rect a] → [Rect a]

widthOfRect (Square s) = s
widthOfRect (Rectangle w _) = w

rectCompare a b =
  widthOfRect a < widthOfRect b

pack rs =
  sort rectCompare (map' rotate rs)
```

For example, packing the list of rects

```
[Square 6, Rectangle 4 3, Rectangle 5 6, Square 2]
```

will give

```
[Square 2, Rectangle 3 4, Rectangle 5 6, Square 6]
```

5

We follow the same pattern as for lists, being careful to deal with exceptional circumstances:

```
seqTake :: (Eq a, Num a) ⇒ a → Sequence b → Maybe (Sequence a)
seqDrop :: (Eq a, Num a) ⇒ a → Sequence b → Maybe (Sequence a)
seqMap :: (a → b) → Sequence a → Sequence b

seqTake 0 _ = Just Nil
seqTake _ Nil = Nothing
seqTake n (Cons x xs) =
  case seqTake (n - 1) xs of
    Nothing -> Nothing
    Just xs' -> Just (Cons x xs')

seqDrop 0 xs = Just xs
seqDrop _ Nil = Nothing
seqDrop n (Cons _ xs) = seqDrop (n - 1) xs

seqMap _ Nil = Nil
seqMap f (Cons x xs) = Cons (f x) (seqMap f xs)
```

6

We can use our `power` function from Chapter 2 Question 4:

```
evaluate :: Expr a → a

data Expr a = Num a
            | Add Expr Expr
            | Subtract Expr Expr
            | Multiply Expr Expr
            | Divide Expr Expr
            | Power Expr Expr deriving Show

evaluate (Num x) = x
evaluate (Add e e') = evaluate e + evaluate e'
evaluate (Multiply e e') = evaluate e - evaluate e'
evaluate (Divide e e') = evaluate e `div` evaluate e'
evaluate (Power e e') = power (evaluate e) (evaluate e')
```

Chapter 11 (Growing Trees)

1

Our function will have type **Eq** a ⇒ a → Tree a → **Bool**. It takes a element to look for, a tree holding that sort of element, and returns `True` if the element is found, or `False` otherwise.

```
treeMember :: Eq a ⇒ a → Tree a → Bool

treeMember x Lf = False
treeMember x (Br y l r) =
  x == y || treeMember x l || treeMember x r
```

Note that we have placed the test `x == y` first of the three to ensure earliest termination upon finding an appropriate element.

2

Our function will have type Tree a → Tree a. A leaf flips to a leaf. A branch has its left and right swapped, and we must recursively flip its left and right sub-trees too.

```
treeFlip :: Tree a → Tree a

treeFlip Lf = Lf
treeFlip (Br x l r) = Br x (treeFlip r) (treeFlip l)
```

3

We can check each part of both trees together. Leaves are considered equal, branches are equal if their left and right sub-trees are equal.

```
equalShape :: Tree a → Tree b → Bool

equalShape Lf Lf = True
equalShape (Br _ l r) (Br _ l2 r2) =
  equalShape l l2 && equalShape r r2
equalShape _ _ = False
```

4

We can use the tree insertion operation repeatedly:

```
treeOfList :: Ord a ⇒ [(a, b)] → Tree (a, b)

treeOfList [] = Lf
treeOfList ((k, v):xs) = treeInsert (treeOfList xs) k v
```

There will be no key clashes, because the argument should already be a dictionary. If it is not, earlier keys are preferred since `treeInsert` replaces existing keys.

5

We can make list dictionaries from both tree dictionaries, append them, and build a new tree from the resultant list.

```
treeUnion :: Ord a ⇒ Tree (a, b) → Tree (a, b) → Tree (a, b)

treeUnion t t' =
  treeOfList (listOfTree t ++ listOfTree t')
```

The combined list may not be a dictionary (because it may have repeated keys), but `treeOfList` will prefer keys encountered earlier. So, we put entries from `t'` after those from `t`.

6

We will use a list for the sub-trees of each branch, with the empty list signifying there are no more i.e. that this is the bottom of the tree. Thus, we only need a single constructor.

```
data Mtree a = Branch a [Mtree a] deriving Show
```

So, now we can define `size`, `total`, and `map` functions:

```
mTreeSize :: Num a ⇒ Mtree b → a
mTreeTotal :: Num a ⇒ Mtree a → a
mTreeMap :: (b → a) → Mtree b → Mtree a

mTreeSize (Branch _ l) = 1 + sum' (map' mTreeSize l)

mTreeTotal (Branch e l) = e + sum' (map' mTreeTotal l)

mTreeMap f (Branch e l) = Branch (f e) (map' (mTreeMap f) l)
```

Chapter 12 (The Other Numbers)

1

We represent points as pairs (2-tuples) of numbers. Since we use `/`, **Fractional** appears in the type.

```
between :: (Fractional a, Fractional b) ⇒ (a, b) → (a, b) → (a, b)

between (x, y) (x', y') =
  ((x + x') / 2, (y + y') / 2)
```

Note, however, that the x-coordinate need not have the same type as the y-coordinate, so long as they are both instances of **Fractional**. Normally, of course, we would expect `a` and `b` to be the same type.

2

The `ceiling` function has type (**RealFrac** a, **Integral** b) ⇒ a → b just like `floor`. We test to see which of the two our number is closest to:

```
roundNum :: (RealFrac a, Integral b) ⇒ a → b

roundNum x =
  if fromIntegral c - x <= x - fromIntegral f then c else f
    where c = ceiling x
          f = floor x
```

We use `fromIntegral` to make sure we can use the real subtraction operator to calculate which number is nearest. We could use `floor (x + 0.5)` instead of `ceiling` if we liked.

Note that it is not quite obvious how rounding should work, with respect to negative numbers, and with respect to numbers which are exactly halfway between integers, such as `3.5`. What rules does our `roundNum` obey? What rules does the built-in Haskell function `round` obey?

3

The whole part is calculated using the built-in `floor` function. We return a tuple, the first number being the whole part, the second being the original number minus the whole part. In the case of a negative number, we must be careful – `floor` always rounds downward, not toward zero!

```
parts :: RealFrac a ⇒ a → (Integer, a)

parts x =
  if x < 0 then
     let (a, b) = parts (- x) in
       (- a, b)
  else
    (floor x, x - fromIntegral (floor x))
```

Notice that we are using the unary negation operator `-` to make the number positive. We use `fromIntegral` to convert the result of the `floor` function into a number of an appropriate type.

4

First we write a function `makeLine` which, given a position, creates a line of spaces, an asterisk, and a newline character:

```
replicate' :: (Eq a, Num a) ⇒ a → b → [b]
makeLine :: (Num a, Eq a) ⇒ a → String

replicate' 0 _ = []
replicate' n x = x : replicate' (n - 1) x

makeLine x =
  replicate' x ' ' ++ ['∗', '\n']
```

It uses the function `replicate'` which makes a list containing many copies of a value. Now we need to determine at which column the asterisk will be printed. It is important to make sure that the range $0 \ldots 1$ is split into fifty equal sized parts, which requires some careful thought. Then, we just print enough spaces to pad the line, add the asterisk, and a newline character.

```
star :: RealFrac a ⇒ a → String

star x =
  let i = floor (x ∗ 50) in
    let i' = if i == 50 then 49 else i in
      makeLine (if i' == 0 then 0 else i' - 1)
```

5

We concatenate the results of all the calls to `star`, which results in the graph:

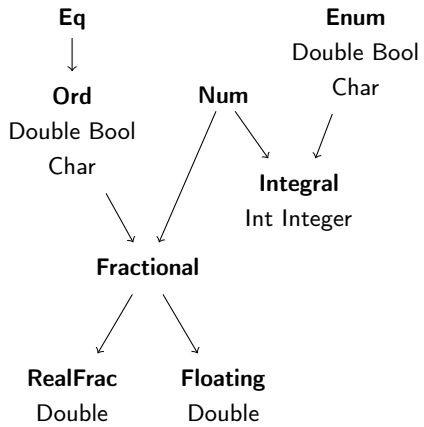
```
plot :: (Ord a, Num a, RealFrac b) ⇒ (a → b) → a → a → a → a → String

plot f a b dy =
  if a > b then
    []
  else
    star (f a) ++ plot f (a + dy) b dy
```

No allowance has been made here for bad arguments (for example, `b` smaller than `a`). Can you extend our program to move the zero-point to the middle of the screen, so that the sine function can be graphed even when its result is less than zero?

6

The types **Bool** and **Char** may be compared for equality and ordered, but they cannot provide the operations necessary to fit into any of the numeric typeclasses. They can, however be enumerated (try `[False ..]` or `['a' .. 'z']`). So we may update the diagram:



Chapter 13 (Being Lazy)

1

This is similar to the `from` function in the text, but we double every time instead of adding one.

```
doubleFrom :: Num a ⇒ a → [a]
doubles :: Num a ⇒ [a]

doubleFrom n = n : doubleFrom (n * 2)

doubles = doubleFrom 1
```

Having written the function which, given a number, doubles from that point, we then just code the list itself by starting at 1. We could use the **`where`** construct to code the list in one shot:

```
doubles :: Num a ⇒ [a]

doubles = doubleFrom 1 where
  doubleFrom n = n : doubleFrom (n * 2)
```

2

This is as simple as using the ++ operator:

```
repeating :: [a] → [a]

repeating l = l ++ repeating l
```

What happens with `repeating []`?

3

The first two Fibonacci numbers are defined to be 0 and 1, so we must give them explicitly. The inner function then puts the first number at the head of the list and shifts everything along one place.

```
fibInner :: Num a ⇒ a → a → [a]
fib :: Num a ⇒ [a]

fibInner x y = x : fibInner y (x + y)

fib = fibInner 0 1
```

Since we are unlikely to need to use `fibInner` outside of `fib`, we can rewrite with the **`where`** constuct:

```
fib :: Num a ⇒ [a]

fib = fibInner 0 1 where
  fibInner x y = x : fibInner y (x + y)
```

4

We can use a tree type with no `Lf` constructor, since our tree will always be infinitely large:

```
data Tree a = Br a (Tree a) (Tree a)
```

We do not use **`deriving`** `Show` because its printed representation will always be infinite. Recall our list version:

```
allFrom :: Num a ⇒ [a] → Tree [a]
allLists :: Num a ⇒ Tree [a]

allFrom l =
  Br l (allFrom (0 : l)) (allFrom (1 : l))

allLists = allFrom []
```

In a similar fashion, we can define one for our new tree type, making use of `interleave` to be sure the left- and right-hand parts of each branch are dealt with fairly:

```
makeList :: Tree a → [a]

makeList (Br x l r) = x : interleave (makeList l) (makeList r)
```

5

We must start with the longest pattern, to avoid any list matching with `x:xs` rather then `x:x':xs`.

```
unleave :: [a] → ([a], [a])

unleave (x : x' : xs) =
  (x : ys, x' : zs) where (ys, zs) = unleave xs
unleave (x : _) = ([x], [])
unleave [] = ([], [])
```

Chapter 14 (In and Out)

1

We can write a helper function `printIntegersInner` which prints the numbers and the commas, being careful to give special attention to the last one in the list. Then, the main function `printIntegers` puts the square brackets on.

```
printIntegersInner :: Show a ⇒ [a] → IO ()
printIntegers :: Show a ⇒ [a] → IO ()

printIntegersInner [] = return ()
printIntegersInner [x] =
  putStr (show x)
printIntegersInner (x:xs) =
  do putStr (show x)
     putStr ","
     printIntegersInner xs

printIntegers l =
  do putStr "["
     printIntegersInner l
     putStr "]"
```

But we notice, from the type, that our function can actually print a list of any showable thing. So its name is somewhat misleading.

2

Our `readThree` **IO** action results in three integers from the user's input. So it has type **IO** (**Integer**, **Integer**, **Integer**). We use the **do** and `<-` constructs together with `getIntegerMaybe` to ask for the three integers in turn, then check that they were valid.

```
readThree :: IO (Integer, Integer, Integer)

readThree =
  do x <- getIntegerMaybe
     y <- getIntegerMaybe
     z <- getIntegerMaybe
     case (x, y, z) of
       (Just a, Just b, Just c) -> return (a, b, c)
       _ ->
         do putStrLn "Not valid integers. Please try again"
            readThree
```

Alternatively, one might read and check each integer individually.

3

Again, we use `getIntegerMaybe`. This time to write a function `readDictNumber` which builds an **IO** action which reads a given number of dictionary entries, making sure to deal with user mistakes. The **IO** action `readDict` deals with asking the user how many entries will be input, again detecting user mistakes.

```
readDictNumber :: Integer → IO [(Integer, String)]
readDict :: IO [(Integer, String)]

readDictNumber n =
  if n == 0 then return [] else
    do i <- getIntegerMaybe
       name <- getLine
       case i of
         Nothing ->
           do putStrLn "Not a valid integer."
              readDictNumber n
         Just x ->
           do rest <- readDictNumber (n - 1)
              return ((x, name) : rest)

readDict =
  do putStrLn "How many dictionary entries to input?"
     n <- getIntegerMaybe
     case n of
       Nothing ->
         do putStrLn "Not a number."
            readDict
       Just i ->
         if i < 0 then
           do putStrLn "Number is negative."
              readDict
         else
           readDictNumber i
```

4

We build a function `row` which, given a file handle and a list of numbers, builds an **IO** action to print out that list separated by spaces. Now, the function `rows` builds a compound **IO** action to print such a list of numbers, multiplied by a given multiplier `n`, using `row` and move to the next line. The main function `table` deals with the tasks of opening and closing the file, and printing out the rows, using multipliers from `1` to `n`.

```
row :: Handle → [Integer] → IO ()
rows :: Handle → Integer → [Integer] → IO ()
table :: FilePath → Integer → IO ()

row fh [] = return ()
row fh (x:xs) =
  do hPutStr fh (show x)
     hPutStr fh "\xs"
     row fh xs

rows fh n [] = return ()
rows fh n (x:xs) =
  do row fh (map (* x) [1 .. n])
     hPutStr fh "\n"
     rows fh n xs

table filename n =
  do fh <- openFile filename WriteMode
     rows fh n [1 .. n]
     hClose fh
```

Another way to write this would be to build a single large string, and then output that at once. There would then only by one function with a type involving **IO**, which we might consider cleaner.

5

The **IO** action built by the `countLinesHandle` function checks to see if we are at the end of the file. Then, there are no more lines, the result is `0`. Otherwise we get one line, count the reset, and add one to the count. The **IO** action built by `countLines` function itself just deals with opening and closing the file.

```
countLinesHandle :: Num a ⇒ Handle → IO a
countLines :: Num a ⇒ FilePath → IO a

countLinesHandle fh =
  do e <- hIsEOF fh
     if e then return 0 else
       do hGetLine fh
          r <- countLinesHandle fh
          return (1 + r)

countLines filename =
  do fh <- openFile filename ReadMode
     lines <- countLinesHandle fh
     hClose fh
     return lines
```

6

The **IO** action built by `copyFileHandle` checks for the end of the input, then copies a line, and calls `copyFileHandle` again. The main function `copyFile` builds an **IO** action which opens the files in the appropriate modes, calls `copyFileHandle`, and closes the files again.

```
copyFileHandle :: Handle → Handle → IO ()
copyFile :: FilePath → FilePath → IO ()

copyFileHandle fromHandle toHandle =
  do e <- hIsEOF fromHandle
     if e then return () else
       do line <- hGetLine fromHandle
          hPutStrLn toHandle line
          copyFileHandle fromHandle toHandle

copyFile fromName toName =
  do fromHandle <- openFile fromName ReadMode
     toHandle <- openFile toName WriteMode
     copyFileHandle fromHandle toHandle
     hClose fromHandle
     hClose toHandle
```

7

Periods, exclamation marks and question marks may appear in multiples, leading to a wrong answer. The number of characters does not include newlines. It is not clear how quotations would be handled. Counting the words by counting spaces is inaccurate – a line with ten words will count only nine.

Chapter 15 (Building Bigger Programs)

1

First, we extend the `Textstat` module to allow frequencies to be counted and expose it through the interface, shown in Figure 15.3. Then the main program is as shown in Figure 15.4.

2

We can write two little functions which build **IO** actions – one to read all the lines from a file, and one to write them. The main **IO** action checks the command line to find the input and output file names, reads the lines from the input, reverses the list of lines, and writes them out. If the command is badly formed, it prints a usage message and exits. This is shown in Figure 15.5.

Note that there is a problem if the file has no final newline – it will end up with one. How might you solve that?

3

See Figure 15.6. We use `linesOfFile` from Question 2, then map over the result to find the length of each line (remember strings are just lists of characters), then sum them. The result is printed to the screen.

```
module Textstat where

import System.IO

data Tree a = Br a (Tree a) (Tree a) | Lf deriving Show

type Stats = (Integer, Integer, Integer, Integer, Tree (Char, Integer))

(length', take', drop', filter', merge, mergeSort, listOfTree, treeLookup, and insert)

updateHistogram ::  (Ord a, Num b) => Tree (a, b) -> [a] -> Tree (a, b)

updateHistogram tr [] = tr
updateHistogram tr (x:xs) =
  case treeLookup tr x of
    Nothing ->
      updateHistogram (insert tr x 1) xs
    Just v ->
      updateHistogram (insert tr x (v + 1)) xs

statsFromChannel :: Handle -> Stats -> IO Stats

statsFromChannel fh (lines, characters, words, sentences, histogram) =
  do ended <- hIsEOF fh
     if ended then
       return (lines, characters, words, sentences, histogram)
     else
       do line <- hGetLine fh
          let charCount = length' line
              wordCount = length' (filter' (\x -> x == ' ') line)
              sentenceCount =
                length'
                  (filter'
                    (\x -> x == '.'|| x == '?'|| x == '!')
                    line)
          statsFromChannel
            fh
            (lines + 1, characters + charCount,
             words + wordCount, sentences + sentenceCount,
             updateHistogram histogram line)

statsFromFile :: FilePath -> IO Stats

statsFromFile filename =
  do fh <- openFile fileName ReadMode
     result <- statsFromChannel fh (0, 0, 0, 0, Lf)
     hClose fh
     return result
```

Figure 15.3: `TextStat.hs`

```
import System.Environment
import Textstat

(put our usual mergeSort and listOfTree functions here)

printHistogramList :: [(Char, Integer)] -> IO ()

printHistogramList [] = return ()
printHistogramList ((k, v):xs) =
  do putStr "For character "
     putStr (show k)
     putStr " the count is "
     putStr (show v)
     putStrLn "."
     printHistogramList xs

printHistogram :: Textstat.Tree (Char, Integer) -> IO ()

printHistogram tree =
  printHistogramList (mergeSort (listOfTree tree))

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile] ->
         do (l, c, w, s, h) <- Textstat.statsFromFile inFile
            putStr "Lines: "
            putStrLn (show l)
            putStr "Characters: "
            putStrLn (show c)
            putStr "Words: "
            putStrLn (show w)
            putStr "Sentences: "
            putStrLn (show s)
       _ -> putStrLn "Usage: Stats <filename>"
```

Figure 15.4: `Stats.hs`

```
import System.Environment
import System.IO

(put our usual reverse' function here)

linesOfFile :: Handle -> IO [String]

linesOfFile fh =
  do finished <- hIsEOF fh
     if finished then return [] else
       do x <- hGetLine fh
          xs <- linesOfFile fh
          return (x : xs)

linesToFile :: Handle -> [String] -> IO ()

linesToFile _ [] = return ()
linesToFile fh (x:xs) =
  do hPutStrLn fh x
     linesToFile fh xs

reverseLines :: FilePath -> FilePath -> IO ()

reverseLines inFile outFile =
  do inHandle <- openFile inFile ReadMode
     outHandle <- openFile outFile WriteMode
     lines <- linesOfFile inHandle
     linesToFile outHandle (reverse' lines)
     hClose inHandle
     hClose outHandle

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile, outFile] -> reverseLines inFile outFile
       _ -> putStrLn "Usage: RevLines input_filename output_filename"
```

Figure 15.5: `RevLines.hs`

```
import System.Environment
import System.IO

(put our usual reverse', map', length' and sum' functions here)

linesOfFile :: Handle -> IO [String]

linesOfFile fh =
  do finished <- hIsEOF fh
     if finished then return [] else
       do x <- hGetLine fh
          xs <- linesOfFile fh
          return (x : xs)

numChars :: Num a => FilePath -> IO a

numChars inFile =
  do inHandle <- openFile inFile ReadMode
     lines <- linesOfFile inHandle
     hClose inHandle
     return (sum' (map' length' lines))

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile] ->
         do size <- numChars inFile
            putStrLn (show size)
       _ ->
         putStrLn "Usage: Size filename"
```

Figure 15.6: `Size.hs`

4

See Figure 15.7. The meat of the solution is in `sortLines`. We must use `read` to read each number as an integer. Since we use read in conjunction with `map'`, the type annotation is on `read` itself, rather than the result of `read`, so it has a function type.

5

See Figure 15.8. It is a simple extension of our earlier function into a standalone program.

6

We can get all the lines in the file using `linesOfFile` from Question 2. The main **IO** action simply uses `matches` on each line, printing the line if `True` is returned.

The interesting function is `matches`. To see if `term` is in `line` we start at the beginning of the string, checking for a match. If a match is not found, we move on one position. This is illustrated in Figure 15.9.

```
import System.Environment
import System.IO

(put our usual reverse', map', merge, mergeSort, take', drop', and length' functions here)

linesOfFile :: Handle -> IO [String]

linesOfFile fh =
  do finished <- hIsEOF fh
     if finished then return [] else
       do x <- hGetLine fh
          xs <- linesOfFile fh
          return (x : xs)

linesToFile :: Handle -> [String] -> IO ()

linesToFile _ [] = return ()
linesToFile fh (x:xs) =
  do hPutStrLn fh x
     linesToFile fh xs

sortLines :: [String] -> [String]

sortLines lines =
  map' show (mergeSort (map' (read :: String -> Integer) lines))

sortNums :: FilePath -> FilePath -> IO ()

sortNums inFile outFile =
  do inHandle <- openFile inFile ReadMode
     outHandle <- openFile outFile WriteMode
     lines <- linesOfFile inHandle
     linesToFile outHandle (sortLines lines)
     hClose inHandle
     hClose outHandle

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile, outFile] ->
         sortNums inFile outFile
       _ ->
         putStrLn "Usage: Size filename"
```

Figure 15.7: `Sort.hs`

```
import System.Environment
import System.IO

copyFileHandle :: Handle -> Handle -> IO ()

copyFileHandle fromHandle toHandle =
  do e <- hIsEOF fromHandle
     if e then return () else
       do line <- hGetLine fromHandle
          hPutStrLn toHandle line
          copyFileHandle fromHandle toHandle

copyFile :: FilePath -> FilePath -> IO ()

copyFile fromName toName =
  do fromHandle <- openFile fromName ReadMode
     toHandle <- openFile toName WriteMode
     copyFileHandle fromHandle toHandle
     hClose fromHandle
     hClose toHandle

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile, outFile] -> copyFile inFile outFile
       _ -> putStrLn "Usage: CopyFile in out"
```

Figure 15.8: `CopyFile.hs`

```
import System.Environment
import System.IO
(put our usual reverse' and filter' functions here)

linesOfFile :: Handle -> IO [String]

linesOfFile fh =
  do finished <- hIsEOF fh
     if finished then return [] else
       do x <- hGetLine fh
          xs <- linesOfFile fh
          return (x : xs)

matches1 :: String -> String -> Bool

matches1 [] _ = True
matches1 _ [] = False
matches1 (x:xs) (y:ys) = x == y && matches1 xs ys

matches :: String -> String -> Bool

matches [] [] = True
matches _ [] = False
matches term (x:xs) = matches1 term (x : xs) || matches term xs

printStrings :: [String] -> IO ()

printStrings [] = return ()
printStrings (x:xs) =
  do putStrLn x
     printStrings xs

search :: FilePath -> String -> IO ()

search inFile searchString =
  do inHandle <- openFile inFile ReadMode
     lines <- linesOfFile inHandle
     let matched = filter' (matches searchString) lines
     printStrings matched
     hClose inHandle

main :: IO ()

main =
  do args <- getArgs
     case args of
       [inFile, searchString] -> search inFile searchString
       _ -> putStrLn "Usage: Search <file name> <search string>"
```

Figure 15.9: `Search.hs`

Chapter 16 (The Standard Prelude and Base)

1

The `Data.Char` module contains the function `toLower` which looks at a character and, if it is upper case, converts it to lower case. Otherwise the character is unaltered. We need only use `map` from the Standard Prelude to complete our tiny program:

```
process :: String → String

process s = map toLower s
```

Of course, we can simplify even further:

```
process :: String → String

process = map toLower
```

2

It is important to get the conditions exactly right.

```
isolate :: (Ord a, Num a) ⇒ [a] → [a]

isolate l =
  takeWhile (> 0) (dropWhile (<= 0) l)
```

Can you extend the function to return a list of all such positive series in a given list?

3

The word **`otherwise`** is simply defined as `True` in the Standard Prelude. Using **`otherwise`** simply makes our programs a little more pleasant to read.

4

The `Data.String` module contains the function `words` which splits a string into words, and `unwords` which does the reverse.

```
f :: String → String

f s = unwords (reverse (words s))
```

Here it is using the `.` operator:

```
f :: String → String

f = unwords . reverse . words
```

5

We can deduce that the numerical difference between each capital and lower case letter is 32:

```
GHCi:
Prelude> import Data.Char
Prelude Data.Char> ord 'a' - ord 'A'
32
```

So we may write our function, here using guarded equations:

```
toLower :: Char → Char

toLower c | c >= 'A' && c <= 'Z' = chr (ord c + 32)
          | otherwise = c
```

The number 32 seems like magic, though, and is harder to understand. We can use our earlier calculation directly:

```
toLower :: Char → Char

toLower c | c >= 'A' && c <= 'Z' = chr (ord c + ord 'a' - ord 'A')
          | otherwise = c
```

Hints for Questions

Chapter 1
Starting Off

1

Try to work these out on paper, and then check by typing them in. Can you show possible steps of evaluation for each expression?

3

Type it in. What does Haskell print? Consider the precedence of + and $*$.

5

What if a value of 2 appeared? How might we interpret it?

Chapter 2
Names and Functions

1

The function takes a number, and returns that number multiplied by ten. So what must its type be?

2

What does the function take as arguments? What is the type of its result? So what is the whole type? You can use the `/=` and `&&` operators here.

3

What is the sum of all the integers from $1 \ldots 1$? Perhaps this is a good starting point.

4

This will be a recursive function. What happens when you raise a number to the power 0? What about the power 1? What about a higher power?

5

Can you define this in terms of the `isVowel` function we have already written?

6

Try adding parentheses to the expression in a way which does not change its meaning. Does this make it easier to understand?

7

When does it not terminate? Can you add a check to see when it might happen, and return 0 instead?

8

Which are expressions and which are function definitions? How many arguments does each function have? What typeclasses must each type variable belong to?

11

Remember that two types are equal if the type variables in one can be renamed to make the other.

Chapter 3
Case by Case

1

We are pattern matching on a boolean value, so there are just two cases: `True` and `False`.

2

Convert the **`if ... then ... else`** structure of the `sum'` function from the previous chapter into a pattern matching structure.

3

You will need three cases as before – when the power is 0, 1 or greater than 1 – but now in the form of a pattern match.

4

For the guarded equation version of `power`, we can begin `powerMatch x n | n == ...`

5

Remember that characters may be compared using the comparison operators.

Chapter 4
Making Lists

1

Consider three cases: (1) the argument list is empty, (2) the argument list has one element, (3) the argument list has more than one element. In the last case, which element do we need to miss out?

2

The function will have type **`Num`** `a` ⇒ [**`Bool`**] → `a`. Consider the empty list, the list with `True` as its head, and the list with `False` as its head. Count one for each `True` and zero for each `False`.

3

To detect if a list is a palindrome, consider the definition of a palindrome – a list which equals its own reverse.

4

Consider the cases (1) the empty list, (2) the list with one element, and (3) the list with more than one element.

5

Can any element exist in the empty list? If the list is not empty, it must have a head and a tail. What is the answer if the element we are looking for is equal to the head? What do we do if it is not?

6

The empty list is already a set. If we have a head and a tail, what does it tell us to find out if the head exists within the tail?

7

Consider in which order the `++` operators are evaluated in the reverse function. How long does each append take? How many are there?

9

Remember that `x `rem` y == 0` if `y` is a factor of `x`.

10

The counting is not done as part of the list comprehension itself.

Chapter 5
Sorting Things

1

Consider adding another **`let`** before **`let`** `left` and **`let`** `right`.

2

Consider the situations in which `take'` and `drop'` can fail, and what arguments `mergeSort` gives them at each recursion.

3

This is a simple change – consider the comparison operator itself.

4

What will the type of the function be? Lists of length zero and one are already sorted – so these will be the base cases. What do we do when there is more than one element?

6

The **`let`** and **`where`** constructs can be used to introduce subsidiary functions, just as they can introduce other expressions.

Chapter 6
Functions upon Functions upon Functions

1

Recall that strings are really lists of characters. So the function `calm` is simple recursion on lists. There are three cases – the empty list, a list beginning with `'!'` and a list beginning with any other character. In the second part of the question, write a function `calmChar` which processes a single character. You can then use `map'` to define a new version of `calm`.

2

This is the same process as Question 1.

3

Look back at the section on anonymous functions. How can `clip` be expressed as an anonymous function? So, how can we use it with `map'`?

4

We want a function of the form `apply f n x = ...` which applies `f` to `x` a total of `n` times. What is the base case? What do we do in that case? What otherwise?

5

You will need to add the extra function as an argument to both `insert` and `sort` and use it in place of the `<=` operator in `insert`.

6

There are three possibilities: the argument list is empty, `True` is returned when its head is given to the function `f`, or `False` is returned when its head is given to the function `f`.

7

If the input list is empty, the result is trivially true – there cannot possibly be any elements for which the function does not hold. If not, it must hold for the first one, and for all the others by recursion.

8

You can use `map'` on each [a] in the [[a]].

9

Use the `filter'` function from Question 6, in addition to functions which reverse and sort lists.

Chapter 7
When Things Go Wrong

1

Make sure to consider the case of the empty list, where there is no smallest positive element, and also the non-empty list containing entirely zero or negative numbers.

2

We can pattern-match on the constructors `Just` and `Nothing` just like any other type.

3

First, write a function to find the number less than or equal to the square root of its argument. Now wrap up your function in another which, on a bad argument, gives zero or otherwise calls your first function. You can use the **`where`** construct to do this.

4

The type will be (a → **Maybe** b) → b → [a] → [b].

5

The type will be (a → **Either** b c) → [a] → ([b], [c]).

Chapter 8
Looking Things Up

1

The keys in a dictionary are unique – does remembering that fact help you?

2

The type will be the similar to (not the same as) the `add` function, but we only replace something if we find it there – when do we know we will not find it? What do we do if we cannot find it?

3

The function takes a list of keys and a list of values, and returns a dictionary. So it will have type [a] → [b] → [(a, b)]. Try matching on both lists at once – what are the cases?

4

This function takes a list of pairs and produces a pair of lists. So its type must be [(a, b)] → ([a], [b]).

For the base case (the empty dictionary), we can see that the result should be `([], [])`. But what to do in the case we have `(k, v):xs`? We must assign names for the two parts of the result of our function on `xs`, and then cons `k` and `v` on to them – can you think of how to do that? Perhaps using the **`where`** construct?

5

You can keep a list of the keys which have already been seen, and use the `elem'` function to make sure you do not add to the result list a key-value pair whose key has already been included.

6

The function will take two dictionaries, and return another – so you should be able to write down its type easily.

Try pattern matching on the first list – when it is empty, the answer is simple – what about when it has a head and a tail?

Chapter 9
More with Functions

2

Try building a list of booleans, each representing the result of `elem'` on a list.

3

The type of `map'` is (a → b) → [a] → [b]. The type of `mapl` is (a → b) → [[a]] → [[b]]. So, what must the type of `mapll` be? Now, look at our definition of `mapl` – how can we extend it to lists of lists of lists?

4

Use our `take'` function to process a single list. You may then use `map'` with this (partially applied) function to build the `truncateLists` function.

5

Build a function `firstElt` which, given the number and a list, returns the first element or that number. You can then use this function (partially applied)

together with `map'` to build the main `firstElts` function.

6

Use `map'` to operate over each list in the list of lists.

Chapter 10
New Kinds of Data

1

The type will have two constructors: one for squares, requiring only a single number, and one for rectangles, requiring two: one for the width and one for the height. You will use the type variable `a` to represent the 'number', just like in the Colour type in the chapter text.

2

The function will have type **Num** a ⇒ Rect a → a. Work by pattern matching on the two constructors of your type.

3

Work by pattern matching on your type. What happens to a square. What to a rectangle?

4

First, we need to rotate the rectangles as needed – you have already written something for this. Then, we need to sort them according to width. Can you use our `sort` function from Chapter 6 Question 5, which takes a custom comparison function?

5

Look at how we re-wrote `length'` and `append` for the Sequence type.

6

Add another constructor, and amend `evaluate` as necessary. You can use the `power` function from Chapter 2, Question 4.

Chapter 11
Growing Trees

1

The type will be **Eq** a ⇒ a → Tree a → **Bool**. That is, it takes an element to search for, and a tree containing elements of the same type, and returns `True` if the element is found, and `False` if not. What happens if the tree is a leaf? What if it is a branch?

2

The function will have type Tree a → Tree a. What happens to a leaf? What must happen to a branch and its sub-trees?

3

If the two trees are both `Lf`, they have the same shape. What if they are both branches? What if one is a branch and the other a leaf or vice versa?

4

We have already written a function for inserting an element into an existing tree.

5

Try using list dictionaries as an intermediate representation. We already know how to build a tree from a list.

6

Consider using a list of sub-trees for a branch. How can we represent a branch which has no sub-trees?

Chapter 12
The Other Numbers

1

Use a pair to represent each coordinate. Ordinary arithmetic will do the rest.

2

Consider the type of the `ceiling` function. You may need `fromIntegral` here.

3

Consider the built-in function `floor`. What should happen in the case of a negative number? You may need `fromIntegral` here.

4

Calculate the column number for the asterisk carefully. How can it be printed in the correct column?

5

You will need to call the `star` function with an appropriate argument at points between the beginning and end of the range, as determined by the step.

6

Are **Bool** and **Char** in **Ord**? **Eq**? **Enum**? What about the numeric typeclasses?

Chapter 13
Being Lazy

1

This is very similar to `from` in the text.

3

Consider the definition of Fibonacci numbers. Split into a function which builds the list given two numbers, and the construction of the list itself.

4

We can use the tree type with just one constructor, `Br`, because leaves will never be needed.

5

The type of the function will be [a] → ([a], [a]). What are the base cases?

Chapter 14
In and Out

1

You may have to treat the last number in the list specially.

2

The `getIntegerMaybe` **IO** action from the chapter text is helpful here.

3

Ask the user the specify, beforehand, how many dictionary entries they are going to submit.

4

Begin with a function which builds an **IO** action to print a single row.

Chapter 15
Building Bigger Programs

3

Remember that a string is really just a list of characters.

4

The `read` function can be used to extract a number from each line of the file.

Chapter 16
The Standard Prelude and Base

4

The functions `words` and `unwords` are suitable.

5

How can we deduce the 'difference' between numbers assigned to lower case and upper case letters? Remember also that the comparison operators can be used on characters.

Coping with Errors

It is very hard to write even small programs correctly the first time. An unfortunate but inevitable part of programming is the location and fixing of mistakes. Haskell has a range of messages to help you with this process.

Here are descriptions of the common messages Haskell prints when a program cannot be accepted or when running it causes a problem (a so-called "run-time error"). We also describe warnings Haskell prints to alert the programmer to a program which, though it can be accepted for evaluation, might contain mistakes.

Errors before the program is run

These are messages printed when an expression could not be accepted for evaluation, due to being malformed in some way. No evaluation is attempted. You must fix the expression and try again.

Syntax errors

Syntax is the arrangement of letters and words and punctuation to make up sentences. In a programming language, a *syntax error* occurs when the arrangement is invalid and so its meaning cannot be determined. For example, if we use the wrong kind of quotation marks around the `div` operator:

```
Prelude> 1 'div' 2

<interactive>:1:3: error:
    • Syntax error on 'div'
      Perhaps you intended to use TemplateHaskell or TemplateHaskellQuotes
    • In the Template Haskell quotation 'div'
```

Notice that Haskell tries to tell us what we may have done wrong. In this case, the advice is incorrect, and indeed contains lots of words we do not understand. Such is the nature of error reporting in a programming language. The numbers `:1:3:` refer to the line and column number where the error occurs. For interactive programming this is of little help, but when compiling with `ghc` we can use the information to find the position in our text editor.

Reserved identifiers

Sometimes we inadvertently use a word which has a special meaning in Haskell, especially when we do not know the whole language:

```
GHCi:
Prelude> (class, teacher) = ("4b", "Mr Carter")

<interactive> error: parse error on input 'class'
```

These special words are printed in bold in this book. They are called reserved identifiers:

```
case class data default deriving do else
foreign if import in infix infixl infixr
instance let module newtype of then type where
```

Parse errors

This error occurs when Haskell finds that the program text contains things which are not valid words (such as **`if`**, **`let`** etc.) or other basic parts of the language, or when they exist in invalid combinations. These cause a failure to parse – the word parse means to read and understand the program's words. Check carefully and try again.

```
GHCi:
Prelude> 1 +

<interactive> error:
    parse error (possibly incorrect indentation or mismatched brackets)
```

For example, here we did not supply a right hand side for the addition operator.

Layout errors

The *layout rule*, introduced in Chapter 3, governs how Haskell constructs may be arranged spatially to make valid programs. The rule is simple to state, but easy to fall foul of, so the beginner is likely to be dealing with error messages arising from bad layout rather often. Sometimes Haskell correctly identifies the problem as a layout issue:

```
GHCi:
Prelude> :{
Prelude| sign x =
Prelude| if x < 0 then -1 else if x > 0 then 1 else 0
Prelude| :}

<interactive> error:
    parse error (possibly incorrect indentation or mismatched brackets)
```

Sometimes, however, Haskell cannot tell us that layout is the problem. You will get used to suspecting and fixing layout issues as a beginner.

Scope errors

This error occurs when you have mentioned a name which has not been defined (technically "bound to a value", or "in scope"). This might happen if you have mistyped the name.

```
GHCi:
Prelude> x + 1

<interactive> error: Variable not in scope: x
```

If the mistake is just a little one, Haskell may suggest the answer. In this case, `true` was probably not an undefined name, but a mistake for the defined constructor `True`:

```
GHCi:
Prelude> true

<interactive> error:
    • Variable not in scope: true
    • Perhaps you meant data constructor 'True' (imported from Prelude)
```

Data constructors can also be non-existent, causing an error:

```
GHCi:
Prelude> Blue

<interactive> error: Data constructor not in scope: Blue
```

Haskell knows it is an undefined data constructor rather than variable here, because it starts with a capital letter.

Type inference errors

One of the most common errors you will have to deal with as a Haskell programmer. For example:

```
GHCi:
Prelude> 1 + False

<interactive> error:
    • No instance for (Num Bool) arising from a use of '+'
    • In the expression: 1 + False
      In an equation for 'it': it = 1 + False
```

The error tells us that we cannot use a number on one side of the plus sign and a boolean on the other. Notice `Num` and `Bool` next to one another and the `'+'`. Often the error occurs late in the type inference mechanism, and so it can be hard to find the source of the error. Practice is all that helps here.

Sometimes, especially when compiling a program with `ghc`, we give both the type of a function and its definition. Haskell will infer the most general type for the function, and then compare with the type given, which may be more restrictive. If the type is less restrictive, or does not match at all, we see an error, and a suggested fix:

```
Prelude> :{
Prelude| f :: a -> a -> Bool
Prelude| f x y = x > y
Prelude| :}

<interactive> error:
    • No instance for (Ord a) arising from a use of '>'
      Possible fix:
        add (Ord a) to the context of
          the type signature for:
            f :: forall a. a -> a -> Bool
    • In the expression: x > y
      In an equation for 'f': f x y = x > y
```

In this case, the message suggests adding the **Ord** a constraint. The unary minus operator `-` can trip us up too:

```
Prelude> 2 * -x

<interactive> error:
    Precedence parsing error
        cannot mix '*' [infixl 7] and prefix '-' [infixl 6] in the same infix
        expression
```

Here, the solution is to parenthesize as `(-x)`.

Warnings before a program is run

Warnings do not stop an expression being accepted or evaluated. They are printed after an expression is accepted but before the expression is evaluated. Warnings are for occasions where Haskell is concerned you may have made a mistake, even though the expression is not actually malformed. You should check each new warning in a program carefully.

Here is a warning that two pattern match cases overlap (in this case, they both match `0` so the second case cannot possibly be taken):

```
GHCi:
Prelude> :{
Prelude| f x =
Prelude|   case x of
Prelude|     0 -> 1
Prelude|     0 -> 2
Prelude| :}

<interactive> warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In a case alternative: 0 -> ...
```

In addition, an error would occur upon evaluation if we called `f` with argument `1`, because no case matches:

```
GHCi:
Prelude> f 1
*** Exception: <interactive>: Non-exhaustive patterns in case
```

The warning for this is not enabled by default, but starting `ghci` with argument `-Wincomplete-patterns` (or just `-W` to enable all warnings) adds this check:

```
<interactive> warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In a case alternative:
        Patterns not matched: a where a is not one of {0}
```

Errors when the program is run

In any programming language powerful enough to be of use, some errors cannot be detected before attempting evaluation of an expression (until "run-time"). For example, asking for the head of an empty list using the Standard Prelude's `head` function:

```
GHCi:
Prelude> head []
*** Exception: Prelude.head: empty list
```

We also saw such 'exceptions' when using `read`:

```
GHCi:
Prelude> read "16.5"
*** Exception: Prelude.read: no parse
Prelude> read "16.5" :: Integer
*** Exception: Prelude.read: no parse
```

These errors should be avoided by using functions which return types such as **Maybe** or **Either** like the `readMaybe` function.

Index