1 Function and Lists

1.1 Programs as functions

At one level a program is a mapping from its input to its output. Almost everything we know from mathematics about functions is almost what we need to know about programs. Some functions, for essential logical reasons, cannot be programs; and in any sufficiently interesting programming language some programs cannot be total functions. (A total function is one which as a defined result for all arguments.)

However, apart from efficiency, two programs are equivalent exactly if they implement the same function. Functions are equal exactly when they give the same results when applied to the same arguments.

1.2 Types

Every Haskell function has a type: the notation `f :: X -> Y` asserts that applying f to anything of type X gives you a thing of type Y. For example

```
sin :: Float -> Float
home :: Person -> Address
add :: (Int, Int) -> Int
logBase :: Float -> (Float -> Float)
```

Float is the type of (single precision) floating point numbers, like 3.1415927 or 2.9979245*e*8; *Int* is the type of (limited precision) integers; (X, Y) is the type of ordered pairs of an element of X and an element of Y.

Notice that Haskell type names start with upper case letters; the names of values, by and large, start with lower case letters.

Mathematicians write $f(x)$ and $\sin(\theta)$ for applications. Sometimes one writes $\sin\ \theta$ with no parentheses; in Haskell it is usual not to write the parentheses. But you often need a space: *sintheta* is one name, *sin theta* is the application of *sin* to *theta*. In Haskell `sin pi`, `sin (pi)`, and `sin(pi)` are all ways of writing the application of *sin* to *pi* (and they all evaluate to something that is not quite zero), and `logBase 10 2`, `(logBase 10) 2`, and `logBase (10) (2)` are all ways of writing the same expression whose value is almost exactly $\log_{10} 2$. On the other hand, you do need parentheses in `add(2,3)`, `add (2, 3)` and `sin(pi/2)`

1.3 Sequences (lists)

Just as a function is a piece of data that captures the idea of computation, one of the ways that the idea of repetition is captured is by sequences, in Haskell pricipally lists. A list of type $[T]$ is a sequence of things each of which is of

type T, for example $[3,1,4,1,5,9,2,7]$ is a $[Int]$, consisting of eight Int values. (There is a pun here: the same brackets [and] are used in type expressions, and in value expressions.)

Where there is an idea of a *next* element of type T, the expression $[a..b]$ evaluates to $[a, a+1, a+2, \ldots b]$. There are lists of every type, including list types, so $[[1],[2,3],[4,5,6]] :: [[Int]]$.

One particularly handy idea is the list comprehension, meant to be reminiscent of set comprehension notation in mathematics. The value of a list comprehension is a list of values of the expression to the left of the bar.

The parts after the bar are taken left-to-right: generators introduce new variables which successively take values from lists, and Boolean valued expressions are guards.

```
> map :: (a -> b) -> [a] -> [b]
> map f xs = [ f x | x <- xs ]

> filter :: (a -> Bool) -> [a] -> [a]
> filter p xs = [ x | x <- xs, p x ]

> concat :: [[a]] -> [a]
> concat xss  = [ x | xs <- xss, x <- xs ]
```

Notice that these apply (uniformly) to lists of all types a. (*map*, *filter* and *concat* are standard functions, and although equal these definitions are not the standard ones.)

So for example

```
map abs [-5..5] = [5,4,3,2,1,0,1,2,3,4,5]
filter even [1..10] = [2,4,6,8,10]
concat [[1],[2,3],[4,5,6]] = [1,2,3,4,5,6]
```

There is a standard definition of the type $String = [Char]$ and lists of this type have a handy compact representation:

```
['a'..'z'] = "abcdefghijklmnopqrstuvwxyz"
concat ["al","pha","bet","ic","al","ly"] = "alphabetically"
```

1.4 Script files

Definitions are written in *script files*. In our case, script files have names ending in `.lhs` (for *literate Haskell script*). Lines starting with > (marks known as *Bird tracks*) are read as definitions, and the rest are comments. To prevent accidents caused by typos, there has to be a blank line between program lines and comment lines. (Literate Haskell scripts encourage explanatory commentary, and are considered to be a Good Thing by all Right Thinking People.)

If you really prefer (and you may find your tutor prefers), you can have `.hs` files in which all lines are definition lines, unless explicitly marked as comments by either `{-` and `-}` brackets, or `--` which marks a comment extending to the end of the line.

1.5 Composition of functions

You will almost certainly be familiar with composition of functions: the composition $f \circ g$ of f and g is $(f \circ g)(x) = f(g(x))$ or in Haskell terms

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

Operators whose names are made of symbols, rather than letters, can appear infixed; and the definition means the same as

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

where the parentheses make `(.)` behave like an ordinary name. The composition operator is a perfectly good function

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

The peculiar order of the types is down to the odd convention that we write arguments to the right of functions, but function types have their argument type on the left!

Compostion is associative, $f \cdot (g \cdot h) = (f \cdot g) \cdot h$, so just as one writes sums like $1+2+3+\cdots+n$ without parentheses, it is perfectly OK to write compositions $f \cdot g \cdot h \cdot i \cdot j \cdot k$ without parentheses. This is quite a common program structure.

1.6 Example: most common words in a text

What are the n most common words in a given text?

The input is a text which is a list of characters, containing visible characters like `'B'` and `','`, and blank characters like spaces and newlines (`' '` and `'\n'`). The Haskell type *Char* is the type of characters, and the type of lists whose elements are of type *Char* is $[Char]$.

The output should be something like:

```
hill: 10
a: 4
names: 3
which: 2
```

This output is also a list of characters, in fact it is the string

```
"  hill: 10\n  a: 4\n  names: 3\n  which: 2\n"
```

Applying *putStr* to a string has the effect of outputting its characters in turn.

So we need to design a function

```
mostCommon :: Int -> [Char] -> [Char]
```

The expression *mostCommon n xs* evaluates to a string of this form, for printing. One scheme would be to:

1. convert all the letters in the text to lower case, so as not to distinguish between words such as "The" and "the";
2. break the text up into words, where by definition a word is a maximal length sequence of visible characters;
3. bring all the duplicated words together, by sorting the list of words into alphabetical order;
4. divide the list of words into runs of the same word;
5. replace each run by a single copy with a count of the number of times it occurs;
6. sort these runs into descending order of count;
7. take the first n elements of the sorted list of runs;
8. convert each run into the characters which will represent it, and concatenate these strings into a single string.

Many of these steps are off-the-shelf library functions in Haskell. Their combination is a composition of functions, right to left.

First of all, for clarity, some types:

```
> type Text = [Char]
> type Word = [Char]
> type Run  = (Int, Word)
```

1. We need a function *canonical* :: *Char* $\rightarrow$ *Char* that converts upper case letters to lower case and (design decision) turns everything else into a blank. This function then needs to be applied by

   ```
   map canonical :: Text -> Text
   ```

   character by character to each character in the text.

2. The standard function

   ```
   words :: Text -> [Word]
   ```

breaks a list of characters into a list of words, where a word is itself a list of characters. Words are non-blanks separated by blanks.

3. To sort a list of words, we need a function $sort :: [Word] \to [Word]$. There is a library function

   ```
   sort :: Ord a => [a] -> [a]
   ```

   which will sort lists of any type, provided there is an ordering on that type. Fortunately, the ordering on lists of characters is exactly what we need.

4. To identify the runs we need a function of type $[Word] \to [Run]$ that replaces runs of identical words with a pair: the length of the run, and the common word. There is a library function

   ```
   group :: Eq a => [a] -> [[a]]
   ```

   which divides a list of any type, provided that there is an equality test on that type, into maximal lists of adjacent equal elements.

5. We need a function $codeRun :: [Word] \to Run$ that will take one of these runs to a length/representative pair.

6. The *sort* function will order these runs, but into descending order of count. One thing we could do is to use the standard function *reverse* to reverse the sorted list $reverse \cdot sort$.

7. There is a standard function *take* which returns an initial segment of the list no longer than a chosen length.

8. We need a function $showRun :: Run \to String$ to turn each run into a corresponding list of characters, then

   ```
   concat . map showRun
   ```

   will produce the required string.

So the whole program is

```
> mostCommon :: Int -> Text -> String
> mostCommon n = concat .              -- :: [String] -> String
>                map showRun .         -- :: [Run] -> [String]
>                take n .              -- :: [Run] -> [Run]
>                reverse .             -- :: [Run] -> [Run]
>                sort .                -- :: [Run] -> [Run]
>                map codeRun .         -- :: [[Word]] -> [Run]
>                group .               -- :: [Word] -> [[Word]]
>                sort .                -- :: [Word] -> [Word]
>                words .               -- :: Text -> [Word]
>                map canonical         -- :: Text -> Text
```

and apart from the library functions, all of which are things we *could* write but need not because someone already has, we need only write

```
> canonical :: Char -> Char
> canonical c | isLower c = c
>             | isUpper c = toLower c
>             | otherwise = ' '
```

which uses Haskell conditional equations and library functions to manage characters.

The function

```
> codeRun :: [Word] -> (Int, Word)
> codeRun xs = (length xs, head xs)
```

is only ever applied to a non-empty list of identical elements and returns the length of that list (*length* is a standard function) and a representative element. Here *head* is a standard function that returns the first element of a list.

```
> showRun :: (Int, Word) -> String
> showRun (n,w) =  concat ["  ", w, ": ", show n, "\n"]
```

showRun constructs a *String* containing the word, a numeral representing its frequency, and a newline character.

```
*Main> putStr hill
When the Anglo-saxons invaded Britain it is clear that they
took over many place names as names, without understanding their
meaning.  The evidence is to be found in names like Penhill,
where Old English hyll was added unnecessarily to a word which
represented Old Welsh pann, hill.  A Penhill in Lancashire
developed into Pendle Hill, a name which means hill-hill-hill.
England also has a Torpenhow Hill, or hill-hill-hill-hill.
*Main> putStr (mostCommon 5 hill)
  hill: 10
  a: 4
  names: 3
  which: 2
  to: 2
```

Exercises

1.1 Recall that function application binds more tightly than any other operators. Put in all the parentheses implicit in the expressions

1. `a plus f x + x times y * z`
2. `3 4 + 5 + 6`
3. `2^2^2^2`

It may help to know that `2^2^2^2` evaluates to 65536, as you can check with GHCi.

1.2 Prove that function composition is associative. (Remember that functions are equal precisely when they return the same result whenever applied to the same argument.)

1.3 Suppose that the $+\!\!+$ operator is defined by

```
as ++ bs = concat [as, bs]
```

(This is not the standard definition, but it defines the same function.)

Is this operator associative? Is it commutative? Does it have a unit (identity element)? Does it have a zero?

(e is a unit of $\oplus$ if $e \oplus x = x = x \oplus e$. z is a zero of $\oplus$ if $z \oplus x = z = x \oplus z$.)

You might be able to tell these things without yet being able to prove that you are right.

1.4 Suppose that

```
double :: Integer -> Integer
double x = 2 * x
```

is the function that doubles an integer. What are the values of

```
map double [3,7,4,2]
map (double.double) [3,7,4,2]
map double []
```

You might check your answers on an interpreter.

Suppose that

```
sum :: [ Integer ] -> Integer
```

is a function that adds up all of the elements of its argument. (There is such a standard function.) Which of the following are true, and why?

$$\begin{aligned} sum.mapdouble &= double.sum \\ sum.mapsum &= sum.concat \\ sum.sort &= sum \end{aligned}$$