Programming I: Functional Programming in Haskell
Unassessed Exercises

These exercises are *unassessed* so you do not need to submit them for marking. They are designed to help you master the language, so you should do as many as you can at your own speed.

There are probably more questions on these sheets than you may need in order to get the hang of a particular concept, so feel free to skip over some of the questions. You can always go back to them later if you need to.

Model answers to each set will be handed out throughout the course.

1 Basics

First read the laboratory notes provided, which introduces you to the Department's computers and tells you how to start the Haskell system. We recommend that you use GHCi but Hugs will also work fine. To complete the exercises in this set you need to load some additional character-handling functions from module `Data.Char`. You can do this by typing "`:module +Data.Char`", or just "`:m +Data.Char`" if you're using GHCi (the '+' means add the module to those currently loaded), or "`:also Data.Char`", or just "`:a Data.Char`" if you're using Hugs.

All the problems on this sheet can now be solved by typing expressions at the prompt, e.g.:

```
Prelude Char> type your expression here and press RETURN
```

As you go through each question, read any accompanying notes carefully. Also, most importantly, make up your own problems and try them out too.

An important objective of this first batch of problems is to introduce you to some of Haskell's built-in types and functions (Haskell's so-called "prelude"). Within GHCi or Hugs, you can find out more about any known identifier, including types, functions, classes etc., by typing `:info` identifier (or use `:i` for short), e.g. `:info +`, `:i Int` etc. To get a list of all function names that are in scope at any time type `:browse` in GHCi or `:names` in Hugs.

Try to do Q1–Q4 before Thursday's lecture, but don't panic if you don't have time. Try to get as many done as you can by the start of week 2.

1. Type in and evaluate the following Haskell expressions:

   (a) `2 - 3 - 4`

   (b) `2 - (3 - 4)`

   What does this tell you about the associativity of the subtraction (`-`) operator?

   Remark: a left-associative operator op implies that successive applications are bracketed from the left, so that `a op b op c` is evaluated as `((a op b) op c)`. For a right- associative operator, it would be evaluated as `(a op (b op c))`.

   (c) `100 ‘div‘ 4 ‘div‘ 5`

   Remark: `div` is the prefix integer division (quotient) function, as in `div 12 5` which gives the answer `2`. The back quotes in `‘div‘` turns the prefix function into an infix operator, as in `12 ‘div‘ 5` (try this out). Do not confuse back quotes with the forward quotes that are used to delimit characters. Note also that an infix operator can be turned into a prefix function by enclosing it in parentheses. For example, the expression `(+) 8 5` means the same as `8 + 5`.

   (d) `100 ‘div‘ (4 ‘div‘ 5)`

(e) `2 ^ 3 ^ 2`

What can you say about the associativity of the `^` operator?

(f) `3 - 5 * 4`

(g) `2 ^ 2 * 3`

(h) `2 * 2 ^ 3`

What can you say about the relative precedence of the `-`, `*` and `^` operators?

Remark: An operator's precedence is an integer in the range `0-9` which is used to determine the order in which multiple operator applications are evaluated. If `op1` has a higher precedence (a higher number) than `op2` then `a op1 b op2 c` evaluates to `((a op1 b) op2 c)` and `a op2 b op1 c` evaluates to `(a op2 (b op1 c))`. We say that `op1` binds more tightly than `op2`, as with `*` and `+` respectively, for example. Experiment with the other Haskell operators.

(i) `(3 + 4 - 5) == (3 - 5 + 4)`

(j) `ord 'a'`

(k) `chr (ord 'b' - 1)`

(l) `chr (ord 'X' + (ord 'a' - ord 'A'))`

(m) `'p' /= 'p'`

(n) `'h' <= 'u'`

Remark: Characters in Haskell are members of the built-in `Char` type, which represents *Unicode* – this defines a numerical coding for a vast array of characters, including the familiar characters such as `'a'`, `'b'`, `'Q'`, `'8'`, `':'` and so on. An important subset of Unicode is the 'ASCII' encoding which defines a code for 128 'basic' characters. The ASCII code for a character can be obtained by the Haskell function `ord` (for ordinal value). The inverse of `ord` is called `chr`. See the attached ASCII table and check it out by applying the `ord` function to various characters. The operators `==`, `/=`, `<`, `>`, `<=`, `>=` also work on characters using the ASCII numeric representation in the obvious way, e.g. `'a' < 'b'` gives `True` because `ord 'a' < ord 'b'`.

(o) `sqrt 2 ^ 2`

(p) `sqrt 2 ^ 2 - 2 (try also sqrt 2 ^ 2 == 2)`

Note that `sqrt` is a function that calculates the square root of a given number. What does this tell you about the relative precedence of prefix function application and infix function application?

Remark: if you're using GHCi, the above gives a rather unintuitive answer. The reason is that floating-point arithmetic, i.e. arithmetic with `Floats` and `Doubles`, is an *approximation* to arithmetic with real numbers. Computers use a finite representation for the infinite real line. The associated arithmetic is therefore subject to artefacts

such as rounding error (we don't get exactly the right answer) and overflow (the answer is too big or too small for the computer to represent). You'll cover this in much more detail in another course.

2. Using the Haskell operators `div`, `mod` and `ord`, where necessary, write expressions to find the values of the following:

   (a) The last digit of the number 123

   (b) The penultimate digit of the number 456

   (c) The eighth lower-case letter of the alphabet (i.e. starting from 'a')

3. All of the following expressions are syntactically correct, but have some form of parsing or type error. For each one add the minimum number of parentheses so that the resulting expression is well formed and well typed.

   (a) `not 2 * 3 == 10`

   (b) `3 == 4 == True`

   (c) `if True then 1 else 2 == 3`

   (d) `ord if 3 == 4 then 'a' else 'b' + 1`

   (e) `8 > 2 ^ if 3 == 4 then 2 else 3 == False`

4. The operators `==`, `/=`, `<`, `>`, `<=`, `>=` also work on tuples. To see how, try the following:

   (a) `(1, 2) < (1, 4)`

   (b) `((4, 9), (3, 1)) > ((1, 10), (9, 4))`

   (c) `('a', 5, (3, 'b')) < ('a', 5, (3, 'c'))`

5. Using `div` and `mod` (infix or prefix) write an expression which converts time (an integer called `s`, say), representing the number of seconds since midnight, into a triple of integers representing the number of (whole) hours, minutes and seconds since midnight. Use a `let` expression for the specific case where `s=8473`, *viz.* `let s = 8473 in ...`.

6. A polar coordinate (r, θ) can be converted into a cartesian coordinate using the fact that the `x`-axis displacement is $rcos(\theta)$, and the `y`-axis displacement is $rsin(\theta)$. Write a `let` expression which converts the polar coordinate $(1, \pi/4)$ into its equivalent cartesian coordinate represented by a pair of `Float`s.

   How to do it? Haskell allows you to use *pattern matching* on tuples in qualified expressions. As an example, in the same way that we can write `let x = 5 in x ^ 2`, which gives `x` the value `5` in expression `x ^ 2` (which evaluates to `25`), we can also write something like `let (x, y) = (5, 6) in x * y` which simultaneously gives `x` the value `5` and `y` the value `6` in the expression `x * y` (which evaluates to `30`). Use pattern matching to solve the problem.

7. This question is designed to exercise further your understanding of pattern matching – it's important, so make sure you understand what's going on.

   For each of the following, explain the answer you get. In at least one case you may get a pattern matching error.

   (a) `let (x, y) = (3, 8) in (x, y, 24)`

   (b) `let (x, y) = (1, 2, 3) in x - y`

   (c) `let (x, (y, b)) = (7, (6, True)) in if b then x - y else 0`

   (d) `let p = (6, 5) in let ((a, b), c) = (p, '*') in (b, c)`

   (e) `let p = (True, 1) in (True, p)`

8. The built-in function `quotRem` takes two integer (e.g. `Int`) arguments, `n` and `m`, say and returns the pair `(n `div` m, n `mod` m)`. Use pattern matching and the `quotRem` function to compute `(n `div` 10 * 10 + n `div` 10) * 100 + (n `mod` 10 * 10 + n `mod` 10)` when `n` is 24, *without* duplicating the subexpressions `n `div` 10` and `n `mod` 10`. What does this expression do given an arbitrary $10 \leq n \leq 99$?

9. Using a combination of arithmetic sequence expressions, string expressions and list comprehensions write expressions to compute:

   (a) The characters in the string "As you like it" whose ordinal value is greater than 106.

   (b) The (positive) multiples of 13 less than 1000 that end with the digit 3.

   (c) The "times tables" for the numbers 2 to 12. The list should comprise triples of the form $(m, n, m * n), 2 \leq m, n \leq 12$.

   (d) The list of honour cards (Jack, Queen, King, Ace) in a pack of cards. Each card should be represented by a pair comprising its value and suit. The value of an honour card is the first character of its name ('J', 'Q, 'K', 'A'). The suits Clubs, Diamonds, Hearts, Spades, should be represented by the characters 'C', 'D', 'H', 'S'. For example, ('A','S') is the ace of spades. Hint: notice that the generator lists comprise characters, i.e. they are `String`s.

   (e) The list of all pairs of numbers $(m, n), m < n$, between 1 and 100 inclusive, whose sum is the same as the square of their absolute difference.

2 Functions

For each function you should also include its type signature. As an additional exercise, when you are happy that your function is correct, comment out the type signature and then return to the GHCi/Hugs prompt. The system will automatically work out, i.e. *infer*, the type of your function and you can check this by typing `:type <fun>`, or `:t <fun>` where `<fun>` is the name of your function. In some cases you may be surprised to see that the inferred type is different to the type you originally wrote down. You may be able to work out what it's doing in these cases; if not the later lectures will help to explain.

1. Write a two-argument function `addDigit` which will add a single-digit integer onto the right-hand end of an arbitrary-size integer. For example, `addDigit 123 4` should evaluate to `1234`. Ignore the possibility of integer overflow.

2. Write a function convert which converts a temperature given in degrees Celcius to the equivalent temperature in Fahrenheit. (To convert from Fahrenheit to Celcius the rule is to subtract `32` from the temperature in Fahrenheit and then multiply the result by `5/9`.)

3. Given the type synonym type `Vertex = (Float, Float)` write a function `distance :: Vertex -> Vertex -> Float` that will calculate the distance between two points, each represented as a `Vertex`.

4. Using the `distance` function write a function `triangleArea :: Vertex -> Vertex -> Vertex -> Float` that will calculate the area of the triangle formed by the three given vertices. Note that a triangle whose sides are of length a, b and c has an area given by

$$\sqrt{s(s-a)(s-b)(s-c)}$$

   where $s = (a+b+c)/2$. Use a where clause to define the value of a, b, c and s.

5. Define a function `isPrime :: Int -> Bool` that indicates whether a positive number is prime or not. A number a is prime if it has exactly 2 distinct divisors that are 1 and a.

   Hint: a is prime if and only if $a > 1$ and each number in the interval $[2, \sqrt{a}\,]$ does not divide a.

6. The factorial of a non-negative integer n is denoted as $n!$ and defined as:

$$n \times (n-1) \times (n-2) \times (n-3) \times ... \times 2 \times 1$$

   0! is defined to be 1. Write a function `fact :: Int -> Int` which returns the factorial of a given non-negative integer. Ignore the possibility of integer overflow.

7. The number of arrangements (permutations) of r objects selected from a set of $n \geq r$ objects is denoted as nPr and defined as:

$$n \times (n-1) \times (n-2) \times ... \times (n-r+1) \text{ or } \frac{n!}{(n-r)!}$$

Define a recursive function `perm` such that `perm n r` evaluates nPr for all non-negative integers n, r, $n \geq r$, without using the earlier `fact` function.

8. The number of ways of choosing r objects from a set of $n \geq r$ objects is denoted nCr, or $\binom{n}{r}$, and is defined as

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Define $\binom{n}{r}$ in terms of $\binom{n-1}{r}$. Hence, define a recursive function `choose` such that `choose n r` computes $\binom{n}{r}$ for all non-negative integers n, r, $n \geq r$. Do not use the earlier `fact` or `perm` functions.

9. Write a function `remainder` which computes the remainder after integer division. Implement the division by repeated subtraction, i.e. do not use the predefined and `mod` function.

10. Write a function `quotient` which similarly defines integer division using repeated subtraction. Ignore division by zero. Use guards to distinguish the base and recursive cases.

11. The binary representation of an integer n can be obtained by generating another integer whose digits, in base 10 representation, comprise only 0s and 1s thus:

    (a) If n is less than 2, its binary representation is just n.

    (b) Otherwise, divide n by 2. The remainder gives the last (rightmost) digit of the binary representation, which is either 0 or 1.

    (c) The preceding digits of the binary representation are given by the binary representation of the quotient from the previous step.

    Write a function `binary` that computes the binary representation of a given integer as described. As an example, `binary 19` should produce `10011`. How would you adapt this to compute the representation of an integer in an arbitrary given base $b \geq 2$?

12. Given an integer argument the built-in 'successor" function `succ` returns the next largest integer. For example, `succ 4` returns `5`. The function `pred` returns the next smallest. Using just `succ` and `pred`, i.e. without using `+` or `-`,

(a) Write a recursive function `add` which will add two non-negative numbers.

(b) Write a recursive function `larger` to determine the larger of two positive integers.

Use guards or multiple recursion equations and pattern matching. Remember that there are two parameters in each case, each of which can be either zero or non-zero. Do you need four equations in each case?

13. Write a recursive function `chop` which uses repeated subtraction to define a pair of numbers which represent the first `n-1` digits and the last digit respectively of the decimal represenation of a given number. Note that this is equivalent to the quotient and remainder after division by 10. However, you are not allowed to use `div` or `mod`!

14. Using `chop`, write a function `concatenate` which concatenates the digits of two non-zero integers. For example, `concatenate 123 456` should evaluate to `123456` and `concatenate 123 0` should evaluate to `123`. Estimate the cost of your concatenate function in terms of the number of multiplications and subtractions it performs.

15. The Fibonacci numbers are defined by the recurrence

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2), \quad n > 1 \end{aligned}$$

Write a Haskell function `fib` that, given an integer `n=0, 1, 2, ...` returns the `n`th Fibonacci number by encoding the above recurrence directly.

If you think about the way a call to this function will be evaluated you will notice that there is an enormous degree of redundancy. For a given n, $f(n-2)$ is evaluated twice, $f(n-3)$ three times, $f(n-4)$ five times etc. Notice any pattern?! A much more efficient way to compute the n^{th} Fibonacci number is to use an auxiliary function that includes as arguments the k^{th} and the $(k+1)^{th}$ Fibonacci numbers for some number k. The idea is that the recursive call is made with the $(k+1)^{th}$ and the $(k+2)^{th}$ Fibonacci numbers - can you see how to generate them from the k^{th} and the $(k+1)^{th}$? You will also need to carry a third argument which records either k or the number of remaining calls to make before one of the arguments is the value, $f(n)$, you need. Define an auxiliary function `fib'` which works in this way and redefine the original `fib` function to call `fib'` appropriately.

16. The Golden Ratio, G, is a beautiful number discovered by the ancient Greeks and used to proportion the Parthenon and other ancient buildings. The golden ratio has this property: take a rectangle whose sides (long:short) are in the Golden Ratio. Cut out the largest square you can

and the sides of the rectangle that remains are also in the Golden Ratio. From this you should be able to work out that $G = \frac{1+\sqrt{5}}{2}$. Curiously, it can also be shown that the ratio of consecutive Fibonacci numbers converges on the Golden Ratio. Defining $r_n = \frac{f(n)}{f(n-1)}$, this says:

$$lim_{n\to\infty}\frac{f(n)}{f(n-1)} \;=\; lim_{n\to\infty}r_n \;=\; G$$

where f is the fibonacci function defined above. Inspired by your earlier `fib'` function build a function `goldenRatio` which takes an accuracy threshold value, `e`, of type `Float` and which returns r_n, where `n` is the smallest integer satisfying

$$|r_n - r_{n-1}| < e$$

Hint: you should start with your optimised `fib'` function from the previous question. To save re-calculating the same ratio twice, you should carry round the the ratio of the previous two numbers you have just computed, i.e. r_k for some k, as a third argument. Use a where clause to define the ratio of the two fibonacci numbers you are carrying round. Note that the absolute value of a number can be obtained using Haskell's `abs` function. Also, to perform the division above, you must first *cast* the integers representing the various Fibonacci numbers into floating-point numbers using the built-in function `fromIntegral`. For now you can think of this function as having type `fromIntegral :: Int -> Float`. This isn't quite right, however: the true picture will emerge later in the course.

3 Lists

These questions are split into three groups: Basics, List Comprehensions and Higher-order Functions. You might like to alternate between the Basics and List Comprehensions questions to start with, as they both exercise important ideas. You might even find ways of using list comprehensions among the basic questions; that's fine.

3.1 Basics

By the end of the course you should be familiar with many of the list processing functions in the Haskell prelude and core modules. You should start with the prelude functions listed in the notes, but are strongly encouraged to browse the Haskell prelude and other modules, where you'll find many more. The `Data.List` module is particularly useful, as you'll see below.

1. By inspection (check them if you like by typing them into GHCi/Hugs), determine whether the following are correctly typed. If they are, determine the resulting value. If not explain the cause of the type error.

   (a) `"H" : ['a', 's', 'k', 'e', 'l', 'l']`

   (b) `('o' : ['n']) ++ "going"`

   (c) `"Lug" ++ ('w' : "orm")`

   (d) `if "a" == ['a'] then [] else "two"`

   (e) `let xs = "let xs" in length [xs]`

   (f) `tail "emu" : drop 2 "much"`

   (g) `head ["gasket"]`

   (h) `zip "12" (1 : [2])`

   (i) `tail ((1,1), (2,2), (3,3))`

   (j) `head [1, (2,3)] /= head [(2,3), 1]`

   (k) `length [] + length [[]] + length [[[]]]`

   (l) `null ["", "1", "11", "111"] !! 1)`

   (m) `zip [5, 3, length []] [(True, False, True)]`

   (n) `unzip [('b', 'd'), ('a', 'o'), ('d', 'g')]`

   (o) `and [1 == 1, 'a' == 'a', True /= False]`

   (p) `sum [length "one", [2, 3] !! 0, 4]`

   (q) `minimum [('b', 1), ('a', 2), ('d', 3)]`

   (r) `maximum zip "abc" [1, 2, 3]`

   (s) `concat [tail ["is", "not", "with", "standing"]]`

   (t) `map head [1..n]`

(u) `filter null (map tail ["a", "ab", "abc"])`

(v) `foldr1 (||) (map even [9, 6, 8 , 2])`

(w) `zipWith (:)  "zip" "with"`

(x) `foldr max 0 [0, 1]`

(y) `foldr maximum [0] [[0,1], [3,2]]`

(z) `zipWith (&&) [True] (filter id [True, False])`

2. The operators `==`, `/=`, `<`, `>`, `<=`, `>=` work on lists as well as the otehr numeric types you've seen. The 'dictionary' style (lexicographical) ordering is perhaps more apparent here. Try the following:

   (a) `"Equals" == "Equals"`

   (b) `"dictionary" <= "dictator"`

   (c) `"False" > "Falsehood"`

   (d) `[1, 1, 1] < [2, 2, 2]`

   (e) `[1, 2, 3] < [1, 1, 5]`

   (f) `[(1, "way")] < [(2, "ways"), (3, "ways")]`

   Now write a function `precedes` (equivalent to `(<=)`) which takes two `Strings` and evaluates to `True` if the first is lexicographically smaller than or equal to the the second. Note that you can generalise the type to make it polymorphic, but you need to learn how to restrict the types to those that are *orderable*. we'll cover this later.

3. Write a function `pos` to find the position of a specified `Int` in a list of `Ints`. Assume that the integers are indexed from 0 and that the specified character will always be found). For example, `pos 4 [3, 4, 0, 1]` should evaluate to `1`.

4. Use the built-in function `elem` write a function `twoSame :: [Int] -> Bool` which delivers `True` iff the given list of integers contains at least one duplicate element. What is the complexity of `twoSame` as a function of the number of elements, n say, in the given list?

5. Define a polymorphic function `rev` (equivalent to the built-in `reverse` function) to reverse a list of objects of arbitrary type. For example, `rev "bonk"` evaluates to `"knob"`. You can add an element `x` to the rightmost end of a list `xs` thus: `xs ++ [x]`. What is its complexity, i.e. what is the cost of your function in terms of the number of `:` operations it performs to reverse a list of length n? Now modify your `reverse` function so that it uses repeated applications of `:` to an accumulating parameter (initially `[]`) instead of repeatedly adding to the rightmost end of a list. What is the complexity of the new function?

6. Write a function `substring :: String -> String -> Bool` which returns `True` iff the first given string is a substring of the second, e.g. `substring "sub" "insubordinate"` evaluates to `True` whilst `substring "not" "tonight"` evaluates to `False`. Hint: one way to do this is to first generate the list of all suffixes of the second string and then compare the first string with the appropriate prefix of each suffix. The result is `True` iff at least one of these comparisons succeeds (use the or function).

7. Two anagrams can be considered to define a transposition of the characters of a third string. For instance, the anagrams `"abcde"` and `"eabcd"` define a transposition in which the last character of a 5-character string is moved to the start, the other characters remaining in the same order. Define a function transpose to transpose the characters of a string as specified by two anagrams. `transpose "UVWXYZ" "fedcba" "ecabdf"` should evaluate to `"VXZYWU"`. Hint: use `pos` and `!!`, noting that all three strings are the same length and that each character of the second string is unique and appears exactly once in the third.

8. Define a recursive function `removeWhitespace :: String -> String` that will remove leading whitespace from a given string. Note: the first of these is guaranteed to be non-whitespace – an important precondition for the next function. The whitespace characters include (`' '`, `'\t'`, `'\n'`) and you can use the built-in function `isSpace` to check for them.

9. Write a function `nextWord` which given a string `s` returns a pair consisiting of the next word in `s` and the remaining characters in `s` after the first word has been removed. A precondition is that the first character in the input string is non-whitespace.

10. Using `removeWhitespace` and `nextWord`, write a function `splitUp` (similar to the built-in function `words`) which returns the list of words contained in a given `String`. The words may be separated by more than one whitespace character and there may be leading whitespace as well. Hint: you should only need to use `removeWhitespace` in one place in your function.

11. Define a function `primeFactors :: Int -> [Int]` that generates the list of prime factors of a given integer $n \geq 1$. To compute the prime factors of an integer n, start with the first prime, 2, and divide n by 2 repeatedly whilst the remainder is 0. This delivers zero or more factors that are all 2, e.g.

    ```
    Main> primeFactors (2^8)
    [2,2,2,2,2,2,2,2]
    ```

    as you would expect. You're not done yet, though. You now have do the same again, this time with 3 and the number that remains after repeated division by 2. For example:

```
Main> primeFactors (3^3 * 2^8)
[2,2,2,2,2,2,2,2,3,3,3]
```

Likewise with 4, 5, 6 and so on. As it happens you don't need to consider even numbers larger than 2 as they are certainly not prime, so you could skip 4, 6, 8 and so on, as an optimisation. What about an odd number like 9? It's actually safe to check this even though it's not prime because a multiple of 9 is also a multiple of 3, which *is* prime: the number at hand cannot therefore be a multiple of 9. Note that if a number is prime then it has only itself as a prime factor. Make sure to test your function in this case.

12. The highest common factor of two integers a and b can be expressed in terms of their prime factors. Suppose `ps` represents the list of primes factors of a, and `ps'` similarly for b. The highest common factor of a and b is the product of common prime factors of a and b, i.e. the product of the elements that are common to `ps` and `ps'`. For example, the highest common factor of `300=2*2*3*5*5` and `375=3*5*5*5` is `3*5*5`, because `3`, `5` and `5` are the prime factors that `300` and `375` have in common. Using the `primeFactors` function above, define a function `hcf :: Int -> Int -> Int` that will compute the highest common factor of two given integers. Use the \\ operator from the `Data.List` module (type `import Data.List` at the top of the program). This takes two lists and delivers the elements of the first list that remain when the elements of the second are removed from it.

13. Again using \\ from module List, write *a* function `lcm :: Int -> Int -> Int` that delivers the lowest common multiple of two given integers. Suppose a is the smaller of the two numbers and b the larger. The lowest common multiple of a and b is the product of the prime factors of a (i.e. a itself!) and the prime factors of b that are not included in the prime factors of a. For example the lowest common multiple of `300=2*2*3*5*5` and `375=3*5*5*5` is `300*5 = 1500`.

3.2 List Comprehensions

1. The following function is supposed to find all the bindings for `x` in a table of `(x, y)` pairs:

   ```
   findAll x t = [y | (x, y) <- t]
   ```

   Try it out with the application `findAll 1 [(1,2),(1,3),(4,7)]`. Oops! What's gone wrong? How would you fix it?

2. Suppose you have a search table comprising a list of (key, value) pairs of type `Eq a => [(a, b)]`. Define a function `remove :: Eq a => a -> [(a, b)] -> [(a, b)]` that will remove all bindings for a given item in

a given table, e.g. `remove 5 [(5,3),(4,6),(5,9)]` should give `(4, 6))`. As an exercise, now write this using a filter instead of a list comprehension, viz. `filter p table` where `table` is the search table. What should `p` be in *point-free* notation?

3. Tony Hoare's famous "quicksort" algorithm works by partitioning a (non-empty) list into those elements less than or equal to that at the head and those greater than that at the head. These two lists are then recursively sorted and the results appended together, with the head element sandwiched between the two. Define a Haskell version of quicksort that generates the two sublists using list comprehensions.

4. The built-in function `splitAt` splits a list at a specified index and returns the elements up to (and including) the indexed element and those that follow, in the form of a pair. For example, `splitAt 2 "12345"` returns `("12", "345")`. Recall that the head of a list is at index 0. Using `splitUp` and `!!` define a function `allSplits :: [a] -> [([a], [a])]` that returns the result of splitting a list at all possible points. For example, `allSplits "1234"` should return `[("1","234"), ("12","34"), ("123","4")]`.

5. Define a function `prefixes :: [t] -> [[t]]` that will compute all prefixes of a given list. For example, `prefixes "123"` should return `["1", "12", "123"]`. Hint: notice in the example that `'1'` appears at the head of every element of the result (a `map` maybe?). Use a list comprehension to implement the map.

6. Define a function `substrings :: String -> [String]` that will compute all substrings of a given string. For example, `substrings "123` should generate `["1", "12", "123", "2", "23", "3"]` in some order. you should be able to use a similar trick to that for `prefixes` above.

7. Recall that the operator \\ in the `List` module removes specified elements from a given list, e.g. `[1,2,3,4] \\[1,3]` returns `[2,4]`. Define a function `perms :: [a] -> [[a]]` that generates all permutations of a given list using \\ and a list comprehension. For example, `perms [1,2,3]` should return, in some order, `[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]`.

8. Suppose a graph is defined by a list of its edges, where each edge is specified as a pair of node identifiers. For example, if the node identifiers are (unique) integers, the following represents a circular graph with four nodes: `[(1,2), (2,3), (3,4), (4,1)]`. Define a function `routes :: Int -> Int -> [(Int, Int)] -> [[Int]]` that will compute all routes from a specified node to another in a given *acyclic* graph, i.e. a graph with no cycles. For example, `routes 1 6 [(1,2), (1,3), (2,4), (3,5), (5,6), (3,6)]` should return `[[1,3,5,6], [1,3,6]]`.

Now extend the function so that it works with cyclic graphs. For example, `routes 1 6 [(1,2), (2,3), (3,4), (4,1)]` should return `[]`. In the previous version, you should find your function will loop indefinitely in this case. Try it!

Hint: Define an auxiliary function that takes the list of nodes you have already visited as an additional parameter.

4 Higher-order functions

1. Rewrite the following functions so that they use either `map`, `concatMap`, `filter` or one of the family of `fold` functions. Hint: if you can't spot the trick try replacing an operator with its (prefix) form.

   (a)
   ```
   depunctuate :: String -> String
   depunctuate []
     = []
   depunctuate (c : cs)
     | elem c ".,:" = depunctuate cs
     | otherwise    = c : depunctuate cs
   ```

   (b)
   ```
   makeString :: [Int] -> String
   makeString []
     = []
   makeString (n : ns)
     = chr n : makeString ns
   ```

   (c)
   ```
   enpower :: [Int] -> Int
   enpower [n]
     = n
   enpower (n : ns)
     = enpower ns ^ n
   ```

   (d)
   ```
   revAll :: [[a]] -> [a]
   revAll []
     = []
   revAll (x : xs)
     = reverse x ++ revAll xs
   ```

   (e)
   ```
   -- Built-in reverse in disguise
   rev :: [a] -> [a]
   rev xs
     = rev' xs []
     where
       rev' [] ys
         = ys
       rev' (x : xs) ys
         = rev' xs (x : ys)
   ```

(f)
```
-- Built-in unzip in disguise
dezip :: [(a,b)] -> ([a],[b])
dezip []
  = ([], [])
dezip ((x, y) : ps)
  = (x : xs, y : ys)
  where
    (xs, ys) = dezip ps
```

2. Write a function `allSame :: [Int] -> Bool` which delivers `True` iff all elements of the given list are the same. One way to do this (not necessarily the best) is to ask whether all adjacent elements in the list are the same, i.e. first=second, second=third and so on. If you haven't already done it this way try it out using `and`, `zipWith` and `tail`.

   Hint: what happens if you zip a list with its own tail?

3. The prelude function `scanl` takes a function, a base value and a list and forms the list whose n^{th} element comprises the results of folding the given function into the first n elements of the list, using the base value given. For example, `scanl (+) 0 [1,3,5,7,9]` computes the *partial prefix* sums of the list `[1,3,5,7,9]`, i.e. the list `[0,1,4,9,16,25]`. The function `scanr` is defined similarly. You can also experiment with variants `scanl1` and `scanr1` which omit the base case in the same way as `foldl1` and `foldr1`.

   (a) Use a single application of `scanl` to build the infinite list of factorials `[1,2,6,24,...]`

   (b) Using `scanl`, `map` and `sum`, compute an approximation to e by summing the first five terms in the infinite expansion for e viz.

$$e = \frac{1}{0!} + \frac{1}{1} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$

   (c) What does the expression `let xs = 1 : scanl (+) 1 xs in xs` compute? Try it. Now explain it!

4. Write a function `squash :: (a -> a -> b) -> [a] -> [b]` which applies a given function to adjacent elements of a list, e.g. `squash f [x1, x2, x3, x4]` → `[f x1 x2, f x2 x3, f x3 x4]`. Implement the function 1. Using explicit recursion and pattern matching and 2. In terms of `zipWith`

   Hint: make use of `tail`, as in the same function above.

5. Write a function `converge :: (a -> a -> Bool) -> [a] -> a` which searches for convergence in a given list of values. It should apply its given convergence function to adjacent elements of the list until the function yields `True`, in which case the result is the first of the two convergent values. For example, `converge (==) [1, 2, 3, 4, 5, 5, 5]` should return `5`. If no convergence is found before the list runs out, return the last

element of the list. A precondition is that the list contains at least one element. Using `converge` and `scanl` (twice), define a (constant) function that will compute e to 5 decimal places.

6. Write a function `limit :: (a -> a -> Bool) -> [a] -> [a]` which again checks for convergence but this time returns the list elements up to the point where the convergence function delivers `True`. This is a generalisation of the `takeWhile` function. We know that if $0 \leq r \leq 1$ then

$$\sum_{n=0}^{\infty} r^n = \frac{1}{1-r}$$

   Use this to test your limit function with an expression of the form `sum (limit f (map (r^) [0..])` for some convergence function `f`. If no limit is found return the whole list.

7. Write a function `repeatUntil :: (a -> Bool) -> (a -> a) -> a -> a`, equivalent to the built-in `until` function, that will iteratively apply a given function of type `a -> a` to some initial value of type `a` until the given predicate yields `True`. For example,

```
*Main> repeatUntil (==15) succ 9
15
*Main> repeatUntil ((==10).length) (1:) []
[1,1,1,1,1,1,1,1,1,1]
```

8. The functions `any, all :: (a -> Bool) -> [a] -> Bool` apply a given predicate to each element of a given list. `any` delivers `True` if one or more applications yields `True` and `all` iff every application yields `True`. Given empty lists they deliver `False` and `True` respectively. Exploiting extensionality, define `any` using `or` and `map` and `all` using `and` and `map` so that they are of the form:

```
any p = ...
all p = ...
```

9. By exploiting extensionality, define the function `isElem`, equivalent to the built-in `elem` function, using `(==)` and `any` so that it is in *point-free form*, i.e. of the form:

```
isElem = ...
```

10. Sometimes we would like to compose a single-argument function with a binary function, as in `(succ . (+)) 4 7`, yielding `12`, for example. However, Haskell's `(.)` function will only compose two single-argument functions.

(a) Define a new composition operator `<.>` that will allow compositions of the above form. What is its most general type?

(b) Now try writing `<.>` in point-free form. You should find you can write it using just the symbols '`(`', '`)`' and '`.`'.

Hint: To write expressions involving operators in point-free form it's often useful to express the operators in their prefix form, e.g. `(.)` in the case of composition. You then need to manipulate the resulting expression so that you can cancel the arguments.

(c) Define point-free versions of `any` and `all` in terms of `<.>`, i.e. versions that are of the form:

```
any = ...
all = ...
```

(d) Now do the same, but this time using the 'vanilla' composition operator `(.)`. Try using the same hint above for point-free functions.

11. Using `(.)`, `foldr`, `map` and the identity function `id`, write a function `pipeline` which given a list of functions, each of type `a -> a` will form a pipeline function of type `[a] -> [a]`. In such a pipeline, each function in the original function list is applied in turn to each element of the input (assume the functions are applied from right to left in this case). You can imagine this as being like a conveyor belt system in a factory where goods are assembled in a fixed number of processing steps as they pass down a conveyor belt. Each process performs a part of the assembly and passes the (partially completed) goods on to the next process. Test your function by forming a pipeline from the function list `[(+ 1), (* 2), pred]` with the resulting pipeline being applied to the input list `[1, 2, 3]`. Hint: Notice that if `f :: a -> a` then `map f` is a function of type `[a] -> [a]`.

5 User-defined data types

1. Define a data type `Shape` suitable for representing arbitrary triangles, whose dimensions are specified by the length of the triangle's three sides (all `Floats`), squares, whose dimensions are given by a single `Float`, and circles, each specified by its radius (again a `Float`). Define a function `area :: Shape -> Float` which returns the area of a given Shape. Recall: The area of a triangle with dimensions a, b, c is given by:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = \frac{a+b+c}{2}$.

2. Now augment `Shape` with (convex) polygons, specified by a list of the polygon vertices, i.e. (x, y) coordinates in the Cartesian plane. Augment the area function accordingly. Note: the area of a convex polygon is the area of the triangle formed by its first three vertices plus the area of the polygon remaining when the triangle has been removed from the polygon. The length of the line joining vertex (x, y) with vertex (x', y') is given by:

$$L = \sqrt{(x-x')^2 + (y-y')^2}$$

3. Define a type synonym `Date` suitable for representing calendar dates. Write a function `age` which given the birth date of a person and the current date returns the person's age in years as an `Int`.

4. Here is the tree flattening function from the notes:

```
flatten :: Tree a -> [a]
flatten Empty
  = []
flatten (Node t1 x t2)
  = flatten t1 ++ (x : flatten t2)
```

   Rewrite this to eliminate the `++` so that there is one call to `:` for each element in the tree. Hint: use a helper function with an accumulating parameter.

5. For the following binary tree data type:

```
data Tree = Leaf | Node Tree Tree
          deriving (Eq, Show)
```

   define a function `makeTrees :: Int -> [Tree]` that, given an integer n, will return the list of all binary trees with n nodes (i.e. `Node` constructors).

`makeTrees 0` should thus return `[Leaf]`. Note: the number of such trees is given by the n^{th} *Catalan number*:

$$\frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$$

You can use this to test your solution. Hint: use a list comprehension.

6. Define a polymorphic data type for describing binary trees in which all values are stored at the leaves; there should be no concept of an empty tree. Using this data type:

   (a) Write a function `build` which will construct a *balanced* binary tree from a non-empty list of values by using the built-in function `splitAt`. The resulting tree should have the property that the elements in the leaves, when read left to right, are in the same order as the original list. You could issue an error message if build is applied to an empty list, or assume a precondition that all input lists are non-empty.

   Note: A balanced binary tree has the property that at each internal node the sizes of the left and right subtrees differ by at most 1.

   (b) Write a function `ends` which will convert a binary tree to a list, preserving a left-to-right ordering of the elements at the fringe of the tree. Your functions should obey the rule:

   ```
   ends (build xs) == xs
   ```

   (c) Write a function `swap` which will interchange the subtrees of a binary tree at every level. For a list `xs` how does the value of `ends (swap (build xs))` relate to `xs`?

7. The following questions relate to a generalisation of the binary trees that we've covered in lectures.

   (a) Define a polymorphic data type `Tree a b` which stores values both at the internal nodes and the leaves, and for which there is an `Empty` data constructor for representing empty trees. Arrange it so that values at the nodes may have a different type (type `a`) to those at the leaves (type `b`).

   (b) Define a function `mapT` which maps two functions over a given `Tree`. The first function should be applied to values at the leaves and the second to values at the nodes.

   (c) Define the function `foldT`, a version of `fold` for objects of type `Tree` which takes two functions, one to transform a given leaf value and one to reduce an internal node. You can picture this as a transformation on a tree which replaces the empty tree by a given base case, and the leaf and node constructors with the supplied functions. Define functions in terms of `foldT` to do the following:

i. Count the number of leaves in a given `Tree a b`. An empty tree should not be counted as a leaf.

ii. Sum the values in a given `Tree Int Int`.

iii. Perform a left-to-right *in-order* flattening of a `Tree a a` to yield a `[a]`.

iv. Perform a right-to-left in-order flattening of a tree which delivers the same result as above, but in the reverse order. Do this by modifying your folding functions; do not use a reverse function on the result from above!

v. Evaluate a `Tree (Int -> Int -> Int) Int` where the internal nodes represent binary functions over integers and the leaves represent integer constants. In order to test this, define a tree with `+` at the root, `*` at the root of the left subtree and `-` at the root of the right subtree. Define the leaves to be 2, 4, 15 and 9 respectively, reading from left to right. The answer should be 14.

6 Type classes

1. Consider an enumerated type such as:

```
data Colour = Red | Green | Blue
            deriving (Show)
```

   (a) Derive the member functions of the class `Bounded`, by adding a `deriving` statement (type `:i maxBound` to see its member functions) and write an expression to compute the `minBound` for `colour`, which should yield `Red`.

   (b) Now derive the member functions of the class `Enum` and write expressions that use the member functions of `Enum` to compute:

      i. The colour that comes after `Green`

      ii. The internal numerical representation of `Blue`

      iii. The colour whose internal numerical representation is 0

      iv. A list of all the colours, `[Red, Green, Blue]`, in order

2. Design a data type `Time` which is capable of representing time in two ways: either in 24-hour format (for example 1356) or in conventional "wall clock" hours and minutes, with an additional flag to indicate whether the time is am or pm, e.g. 1:56pm. Use an additional enumerated data type `AmPm` to represent the two possibilities. Use `deriving` to enable times to be both displayed and compared for structural equality. Notice that a comparison of the two times above will yield `False` because they are structurally different, even though they depict the same time.

(a) Write a function `to24` which will convert a time in either format to 24-hour format. A convention is that 12:00pm is midday (1200) and 12:00am is midnight (0000).

(b) Using `to24` write a function `equalTime` which given two times in either format returns `True` if the times are the same; `False` otherwise. For example, given the representations of 1514 and 3:14pm, `equalTime` should return `True`.

(c) Now delete the `deriving` statement from your data type and make `Time` an instance of the class `Eq`. At the same time define `==` on `Time`s which delivers `True` iff two times are equal, regardless of their format. Test your implementation by comparing various `Time`s for both equality and inequality (recall that `/=` is defined by default in terms of `==` in the definition of class `Eq`).

(d) Now make `Time` an instance of the class `Show` and provide a definition of the function `show` for displaying `Time`s. A time in 24-hour format should be displayed in the form `XXXXhrs` where `XXXX` is the four-digit 24-hour time. For a time in the conventional format, hours (`HH`) and minutes (`MM`) should be displayed in the form `HH:MMam` for times before midday, and `HH:MMpm` for times after midday, with the special cases that 12-00pm and 12-00am should be displayed as `"Midday"` and `"Midnight"` respectively. Test your implementation by typing various expressions of type `Time` on the Haskell command line.

3. This is a question with few hints, where you have to work out the details yourself. You're given the following types and operator/function definitions:

```
type VarName = String

data Fun = Add | Sub | Mul
         deriving (Eq, Show)

data Exp = Val Int | Id VarName | App Fun Exp Exp
         deriving (Eq, Show)

type Assignment = (VarName, Exp)

type Program = [Statement]

data Statement = A Assignment | Loop Int Program

type Environment a = [(String, a)]

infixl 1 <--
(<--) :: VarName -> Exp -> Statement
```

```
(<--)
  = (A .) . (,)

loop :: Int -> Program -> Statement
loop = Loop
```

The operator `<--` is going to be used to represent assignment, equivalent to `=` in Java or C. Your mission is to build a set of type classes, class instances and functions that enable you to write simple procedural programs as an *embedded* language in Haskell. For example, the following iembedded program should initialise the variables `x`, `y` and `z` as shown and then update them by performing two iterations of the loop body shown:

```
p1 :: Program
p1 = [x <-- 8,
      y <-- 0,
      z <-- 11,
      loop 2 [x <-- x + z - y,
              z <-- x - y + z,
              y <-- 2 * y - x]]
```

If you define a function `run :: Program -> Environment Int` then the idea is that `run p1` should return an environment that contains the values of the three variables after two iterations of the loop.

How to do it?! We're going to cheat a bit by defining a type class `Vars` that enumerates all the variables we will ever need, e.g. for the above program:

```
class Vars a where
  x, y, z :: a
```

Now, notice how `x` in the input needs to be interpreted as a `VarName` (`String`) on the left of the `<--` and an `Exp` on the right. You therefore need to make both `String` and `Exp` instances of `Vars`. What should those instances look like? Now, you also want `+`, `-` and `*` to be recognized as functions over `Exp`s, so you'll need to make `Exp` an instance of `Num`.

To complete the exercise all you have to do is write a little interpreter for assignments and loops and you're done. For example, define `evalS :: Statement -> Environment Int -> Environment Int` and make `run` call `evalS`.

Important: In order to make `String` an instance of `Vars` you have to turn on the `FlexibleInstances` language extension. This is because Haskell normally requires instance types to be parameterised by zero or more type *variables* and `String` is synonymous with `[Char]`, not, for example, `[a]`. Note that `[a]` is essentially interpreted as `[] a`, where `[]` is treated as a

type *constructor*. To get round this, put the following at the top of your program:

```
{-# LANGUAGE FlexibleInstances #-}
```