Programming I: Functional Programming in Haskell

Unassessed Exercises 1: **Basics** (Solutions)

These exercises are unassessed so you do not need to submit them. 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.

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 GHC. 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`" in GHCi (the '+' means add the module to those currently loaded).

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, 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.

Try to complete as many of these as you can before Thursday's lectures. If you don't finish them all don't panic!

1. This question is about evaluating various Haskell expressions.

   (a) Evaluate the following expressions, what do they tell you about the associativity of the `(-)` subtraction operator?

      i. `2 - 3 - 4`

      ii. `2 - (3 - 4)`

   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))`.

   Solution: The subtraction operator `(-)` is left-associative.

(b) Any function with two (or more) arguments can be written infix by using backticks[1]: `f x y` is the same as ``x `f` y``. What do the results of the following expressions tell you about the associativity of infix function application?

   i. ``100 `div` 4 `div` 5``

   ii. ``100 `div` (4 `div` 5)``

   Remark: `div` is the prefix integer division (quotient) function, as in `div 12 5` which gives the answer `2`. Note also that an infix operator can be turned into a prefix function by enclosing it in parentheses. For example, the expression `(+) 8 5` is the same as `8 + 5`.

   Solution: Infix function application with backticks is left-associative. Note that (ii) will crash with a `*** Exception: divide by zero`.

(c) Evaluate `2 ^ 3 ^ 2`, what can you say about the associativity of the `(^)` operator?

   Solution: Exponentiation is right-associative.

(d) By evaluating the following expressions, what can you say about the relative precedence of the `(-)`, `(*)` and `(^)` operators?

   i. `3 - 5 * 4`

   ii. `2 ^ 2 * 3`

   iii. `2 * 2 ^ 3`

   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.

   Solution: Exponentiation `(^)` binds tighter than multiplication `(*)`, which in turn binds tighter than subtraction `(-)`.

(e) Evaluate `(3 + 4 - 5) == (3 - 5 + 4)`, what is the result?

   Solution: `True`

(f) Evaluate the following expressions, what are their results?

   i. `ord 'a'`

   ii. `chr (ord 'b' - 1)`

   iii. `chr (ord 'X' + (ord 'a' - ord 'A'))`

   iv. `'p' /= 'p'`

   v. `'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

[1]Do not confuse the backtick, `` ` ``, with the quote, `'`, used for characters!

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'`.

(g) The `sqrt` function calculates the square root of a given number. By evaluating the following expressions, what can you say about the relative precedence of prefix function application and infix function/operator application?

i. `sqrt 2 ^ 2`

ii. `sqrt 2 ^ 2 - 2` (try also `sqrt 2 ^ 2 == 2`)

Remark: if you're using GHCi, the above gives a rather unintuitive answer. The reason is that floating-point arithmetic, i.e. arithmetic with `Float`s and `Double`s, 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.

Solution: Prefix function application binds tighter than infix application.

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

Solution: `` 123 `mod` 10 ``

(b) The penultimate digit of the number 456

Solution: `` 456 `div` 10 `mod` 10 ``

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

Solution: `chr (ord 'a' + 7)`

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`

Solution: `not (2 * 3 == 10)`

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

Solution: `(3 == 4) == True`
This is Nick's favourite joke!

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

Solution: `(if True then 1 else 2) == 3`

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

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

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

Solution: `(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 ....`

Solution:

```
let s = 8473
    hours = s `div` 3600
    minutes = s `mod` 3600 `div` 60
    seconds = s `mod` 60
in (hours, minutes, seconds)
```

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.

Solution: `let (r, t) = (1, pi/4) in (r * cos t, r * sin t)`

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)`

   Solution: `(3, 8, 24)`, since `x` and `y` are *bound* to 3 and 8, respectively.

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

   Solution: This produces a pattern match error, the left-hand side is a tuple of *arity* 2, but the right-hand side has arity 3.

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

   Solution: `x` will be assigned to 7 and `y` will become 6, look at the relative structure of the nested tuples!

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

   Solution: The expression `(p, '*')` can be expanded to `((6, 5), '*')` due to the definition of `p`. From there, the logic is the same as part c.

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

   Solution: `(True, (True, 1))`, see above.

   (f) `let (True, x) = (True, 1) in x - 1`

   Solution: As `True` is found on the left-side of the right-hand tuple, this matches, and results in `1-1`

   (g) `let (True, x) = (False, 1) in x - 1`

   Solution: This compiles, but crashes, since `False` is not `True` and this is a pattern match that is not allowed to fail (there are no other alternatives).

   (h) `let (x, y, 0) = (1, 2, 0) in x + y`

   Solution: This works, as `0` is found in both 3rd positions in the tuples: it results in `1+2`.

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 $10 \leq n \leq 99$?

Solution:

```
let (q, r) = quotRem 24 10
in (q * 10 + r) * 100 + (r * 10 + r)
```

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.

   Solution: `[c | c <- "As you like it", ord c > 106]`

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

   Solution: ``[x | x <- [13,26..1000], x `mod` 10 == 3]``

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

   Solution: `[(m, n, m * n) | m <- [2..12], n <- [2.12]]`

   (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'`, and `'S'`, respectively. For example, `('A','S')` is the ace of spades. Hint: notice that the generator lists comprise characters, i.e. they are `String`s.

   Solution: `[(c, s) | c <- "JQKA", s <- "CDHS"]`

   (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.

   Solution:

   ```
   [(m, n) | m <- [1..100]
           , n <- [1..100]
           , m < n
           , (m + n) == (m - n) ^ 2]
   ```