COMP313-08A Programming Languages

First Haskell coursework

Solutions

The first few questions of this coursework are intended to get you started using the hugs interpreter with some simple definitions. There are some larger problems later on in the coursework. I strongly suggest that you read through my "background" notes on Haskell, accessible from the course web site as `http://www.cs.waikato.ac.nz/~stever/Haskell_notes.pdf`, since some of the questions refer to ideas in those notes which we may not yet have covered in the lectures.

1. *What is the type of* `(2 *)`*?*
2. *What value (remember that functions are values) does* `(2 *)` *have?*

1 and 2.

```
(2 *) :: Int -> Int
```

This is so because `(*) :: Int -> Int -> Int,` i.e. `(*)` is a function which, given its first argument, i.e. `2`, has as value another function, of type `Int -> Int`. This function, as you can see from its type, has one integer argument and returns an integer result — the result is the value of its argument multiplied by two.

(In fact, the interpreter will tell you that

```
(2 *) :: Num a => a -> a
```

i.e. that assuming the type `a` is in the class `Num`, then the function has type `a -> a`. Since we haven't (yet) covered this in the course you should NOT have given this answer (did you even know what it means??), however I'll accept either answer.)

The value of this is: a function of one argument that doubles this argument.

Using lambda-expressions (covered soon in the course) we can say that the value of `(2 *)` is:

```
\x -> 2 * x
```

Remember that functions are just values too: there is no distinction between integers, characters, lists of integers or functions. This is something new which takes some getting used to, but once you are used to it you will find it a very elegant and powerful way of thinking.
Remember, functions are first-class citizens!

3. *Create a file called ex1 and type in the following definitions:*

```
twice f x = f (f x)
square n = n * n
naturals = [0..]
ones = 1 : ones
```

and then load this file using the load command. Once the file is loaded, try to answer the following questions:

a. *What is the type of* `twice`*?*
b. *What is the value of* `twice not True`*?*
c. *what is the type of* `square`*?*
d. *what is the type of* `twice square`*?*
e. *what is the value of* `twice square 4`*?*
f. *what is the type of the pre-defined function* `head`*?*
g. *what is the type of* `head naturals`*?*
h. *what is the value of* `head naturals`*?*
i. *draw a picture of the list* `naturals`*.*
j. *do the same for the list* `ones`*.*

a. `(a -> a) -> a -> a`
b. `True`

This is so since

```
    twice not True
--> not (not True)
--> not False
--> True
```

c. . `Int -> Int`

We know that the type of `twice` is `(a -> a) -> a -> a`. In order that `twice square` be well-typed, we have to be able to match the type of `square`, `twice`'s argument, with `a -> a`. This can be done if we substitute `Int` for `a`. Then, remember, having provided `twice` with an argument that fits (as far as the types are concerned) we are left with the type `a -> a`, i.e. the `(a -> a)` goes since we have provided an argument that fits this. Hence we are left with `a -> a`, i.e. `Int -> Int`, for the type of `twice square`.

e. `256`

f. `[a] -> a`

g. `Int`

h. `0`

i. `0 ----> 1 -----> 2 -----> 3 -----> 4 -------> 5 -------> 6 -----`
`---> 7 -------->` ……

and so on for as long as you can be bothered.

j. `1 --------> 1 ---------> 1 ----------> 1 -----------> 1 --------`
`-> 1 ------->` …….

and so on, again, for as long as you can be bothered.

4. *What expression, using* *<u>only</u>* *the function* `square` *and the number* *2, has the value* *16?*

```
square (square 2)
```

This is so since

```
    square (square 2)
--> (square 2) * (square 2)
--> (2 * 2) * (square 2)
--> 4 * (square 2)
--> 4 * (2 * 2)
--> 4 * 4
--> 16
```

5. *What expression, using* *<u>only</u>* *the list* `naturals`*, the function* *`(2 *)`* *and the function* `map`*, represents the list of all even natural numbers?*

```
map (2 *) naturals
```

This is so since

```
map (2 *) naturals
--> map (2 *) [0..]
--> (2 *) 0 : map (2 *) [1..]
--> (2 *) 0 : (2 *) 1 : map (2 *) [2..]
--> (2 *) 0 : (2 *) 1 : (2 *) 2 : map (2 *) [3..]
--> ...
```

and this can be carried on for as long as you like. If you now start to do some arithmetic, perhaps forced by wanting to print parts of the list, then you see that the first element of this list, `(2 *) 0` = 0, the second is `(2 *) 1` = 2, the third is `(2 *) 2` = 4 and so on. Clearly we have as much as we want of the list of all the even natural numbers, as required.

6. *Define a function*

```
capitalize :: Char -> Char
```

which given any alphabetic character returns its capital form. On numbers and punctuation it should leave the character alone.

You will find the definitions

```
ord :: Char -> Int
ord = fromEnum

chr :: Int -> Char
chr = toEnum
```

useful. So,

```
capitalize 'a' = 'A'
capitalize 'e' = 'E'
capitalize 'D' = 'D'
capitalize '{' = '{'
capitalize '3' = '3'
```

By experimenting with `ord` and `chr` you can work out the codes for lower and upper-case letters. This gives the offset in the arithmetic in the following:

```
capitalize c
       | c >= 'a' && c <= 'z' = chr (ord c - 32)
       | otherwise = c
```

7. Using only the pre-defined function `map` and the function `capitalize` you gave in question eight, define a function

```
capitalize_string :: String -> String
```

which takes a string and applies `capitalize` to each element. So,

```
capitalize_string "Hello, Steve" = "HELLO, STEVE"
capitalize_string "Happy Birthday on the 31st" =

                                 "HAPPY BIRTHDAY ON THE 31ST"
```

```
capitalize_string = map capitalize
```

8. Using only `foldr`, the Boolean operation `(||)`and `False`, define a function

```
or_list :: [Bool] -> Bool
```

so that

```
or_list [b1, b2,...,bn] = b1 || b2 ||...|| bn
```

Note that

```
or_list [] = False
```

```
or_list = foldr (||) False
```

9.
i. Define, using only a list comprehension, the list `[1..k]` for some `k`, the string `"sheep\n"`and the pre-defined function `concat`, a function

```
flock :: Int -> String
```

which, when given a number, say five, as argument, is such that

```
putStr (flock 5)
```

gives

```
sheep
sheep
sheep
sheep
sheep
```

i.e. five sheep, one on each line, as value.

```
flock n = concat ["sheep\n" | x <- [1..n]]
```

ii. Now define

```
flock2 :: Int -> String
```

which does the same thing but WITHOUT using a list comprehension, but using anything else you like.

```
flock2 0     = ""
flock2 (n+1) = "sheep\n" ++ flock2 n
```

iii. Using <u>only</u> a list comprehension, the string `"sheep "`*, lists of natural numbers and the pre-defined function* `concat`*, define the function* `a_row_of_sheep :: Int -> String`

so that `a_row_of_sheep 5`*, for example, has value*

```
sheep sheep sheep sheep sheep
```

```
a_row_of_sheep n = concat ["sheep " | y <- [1..n]]
```

iv. By considering your answers to i and iii and generalizing, define a function

```
big_flock :: Int -> String
```

so that `putStr (big_flock 5)`*, for example, has value*

```
sheep
sheep sheep
sheep sheep sheep
sheep sheep sheep sheep
sheep sheep sheep sheep sheep
```

```
big_flock n = concat [ a_row_of_sheep x ++ "\n" | x <- [1..n]]
```

10.
For this question do all of Exercise 2.5 on page 33 of the textbook.

The method I've used is the obvious one: traverse the list of vertices, and if the first x co- ordinate is not less than the next x co-ordinate then add the relevant trapezium, otherwise if the first x co-ordinate is less than the next one, subtract the relevant trapezium.

```
-- First set things up by adding the first vertex again as
-- the last vertex so we have a closed polygon

area' (Polygon (v0:v1:v2:vs)) =
                         newarea (Polygon ((v0:v1:v2:vs) ++ [v0]))

-- Don't bother working out area for fewer than three
-- vertices, because it's not a polygon then!

area' _ = 0

-- The obvious definition (and "area" is the original area
-- function from the Shape module

newarea (Polygon ((x0,y0) : (x1,y1) : vs))
   | x1 >= x0 =   area (Polygon [(x0,y0),(x1,y1),(x1,0),(x0,0)]) +
                                    newarea (Polygon ((x1,y1):vs))
   | x1 <= x0 = - area (Polygon [(x0,y0),(x1,y1),(x1,0),(x0,0)]) +
                                    newarea (Polygon ((x1,y1):vs))

newarea (Polygon _) = 0

-- A better solution: same method, but use "where" to avoid
-- repeating calculations....

newarea (Polygon ((x0,y0) : (x1,y1) : vs))
   | x1 >= x0 =   trap_area + rest
   | x1 <= x0 = - trap_area + rest
where trap_area = area (Polygon [(x0,y0),(x1,y1),(x1,0),(x0,0)])
           rest = newarea (Polygon ((x1,y1):vs))

newarea (Polygon _) = 0
```

Your answers for all the questions above are due at 1000 on Wednesday 26th March 2008.

*You must submit your answers as a plain text file via Moodle. This **MUST** be a plain text file (**not** a PDF, **not** a MS-format file or any other sort of particular format) since we will want to load your solutions and try them out in hugs.*