1 Lab 5 Haskell

Date: Apr 22, 2021

This document first describes the aims of this lab followed by exercises which need to be performed.

You should perform all exercises using the `hugs` dialect of Prolog which is installed on remote.cs.

1.1 Aims

The aim of this lab is to introduce you to Haskell. After completing this lab, you should have some familiarity with the following topics:

- Haskell types and function application.
- Using Haskell's pattern matching to access components of data structures.
- List comprehensions in Haskell.
- Using map and filter.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab5` branch and a new `submit/lab5` directory.

For this lab, all code should be written in a single file `lab5-sol.hs` residing in your `submit/lab5` directory. Submit that file along with a log of your terminal interaction.

Note that you can repeately load that file into your Haskell interpreter using:

```
Hugs> :l "lab5-sol.hs"
```

If already loaded, you can reload by simply using:

```
Hugs> :r
```

1.2.2 Exercise 1: Haskell Types and Function Application

All Haskell expressions are statically typed, but usually type declarations are optional since Haskell infers types automatically. Within the Hugs REPL, you can use the `:type` directive (abbreviated `:t`) to check the type of an expression.

Add the following to your `lab5-sol.pro` file:

```
-- Exercise 1

-- function which adds two numbers
add n1 n2 = n1 + n2

-- same, but define without using args
plus = (+)

-- function which concats two lists
conc ls1 ls2 = ls1 ++ ls2
```

Load the `lab5-sol.hs` file into the REPL:

```
Main> :l "lab5-sol.hs"
Main> :t add
Main> :t plus
Main> :t conc
```

expr `::` *Type* should be read as *expr* has type *Type*; The LHS of the `=>` operator is used to qualify the type variables used on its RHS. A type expression like `a -> b` should be read as a function from `a` to `b`; `->` is right-associative, i.e. `a -> b -> c` associates as `a -> (b -> c)`.

Now type in the following to use the above functions:

```
Main> add 2 3
Main> conc [1] [2, 3]
Main> conc [[1]] [[2, 3]]
Main> conc "hello" "world"
Main> conc ["hello"] ["world"]
Main> conc (conc ["hello"] ["world"]) ["goodbye"]
Main> conc (conc ["hello"] ["world"]) [42]
```

- What happens if the parentheses are omitted on the second-last line above? Why?
- Why does the last line result in an error?

Add the following function definitions to your `lab5-sol.hs` file:

```
-- partially apply above functions:

add10 = add 10
plus5 = plus 5
concHello = conc "hello"
```

Reload your Haskell REPL with the above file and type directives to look at the types of these new functions. Also type expressions which use these new functions to compute results.

1.2.3 Exercise 2: Haskell Pattern Matching

Haskell supports n-tuples $(a_1, \ldots, a_n)$ when $a_1, \ldots, a_n$ can have different types. A 2-tuple is known as a **pair** and a 3-tuple is a **triple**. Haskell has built-in functions `fst` and `snd` to extract the first and second component of a pair:

Type the following into your REPL:

```
Main> let tuple = ("hello", 42) in fst tuple
Main> let tuple = ("hello", 42) in snd tuple
```

We can define our own versions of `fst` and `snd` in `lab5-sol.hs`:

```
-- Exercise 2

first (v, _) = v
second (_, v) = v
```

Reload your `lab5-sol.hs` into the REPL and rerun the `fst` and `snd` examples using `first` and `second`.

1. If you try something like `first (12, "hello", [])` you will get an error. Fix this by defining `fst3` and `snd3` functions which work on triples.
2. Recall that the `:` infix operator is Haskell's equivalent of Scheme's `cons`. Use pattern matching to define a function:

   ```
   sumFirst2 :: Num a => [a] -> a
   ```

   which returns the sum of the first 2 elements of a list of numbers.
3. Use pattern matching to define a function

   ```
   fnFirst2 :: [a] -> (a -> a -> b) -> (a -> a -> b) -> b
   ```

   which takes a list `ls` and two functions `f1` and `f2` as arguments. If the list has exactly two elements, then the result of the function should be `f1` applied to the first two elements of `ls`; otherwise it should be `f2` applied to the first two elements of `ls`.

   ```
   Main> fnFirst2 [3, 4] (+) (*)
   7
   Main> fnFirst2 [3, 4, 5] (+) (*)
   12
   ```

1.2.4 Exercise 3: List Comprehensions

Add the following to your `lab5-sol.hs` file:

```
-- Exercise 3

cartesianProduct ls1 ls2 =
  [ (x, y) | x <- ls1, y <- ls2 ]

cartesianProductIf ls1 ls2 predicate =
  [ (x, y) | x <- ls1, y <- ls2, predicate x y ]
```

Then run

```
Main> cartesianProduct [1..4] [2..4]
Main> cartesianProductIf [1..4] [2..4] (>)
```

and understand how list comprehensions work.

1. Within the REPL, write a list comprehension which produces the list of pairs (x, y) such that $y = 3x^2 + 2x + 1$ for $x \in 1, 2, \ldots, 10$. Recall that `[1..10]` will produce the list `[1, 2, ..., 10]` and `x^2` will return the square of `x`.

   The result should be:

   ```
   [(1,6),(2,17),(3,34),(4,57),(5,86),(6,121),(7,162),
    (8,209),(9,262),(10,321)]
   ```

2. Repeat the previous exercise but return only those pairs whose second components are a multiple of 3. **Hint**: The Haskell function `rem m n` returns the remainder after dividing `m` by `n`.

   The result should be:

   ```
   [(1,6),(4,57),(7,162),(10,321)]
   ```

3. In `lab5-sol.hs`, write a function `oddEvenPairs n` which returns pairs (x, y) such that $1 \leq x, y \leq n$ and x is odd while y is even. **Hint**: Use Haskell's `odd`, `even` built-in predicates.

   ```
   Main> oddEvenPairs 5
   [(1,2),(1,4),(3,2),(3,4),(5,2),(5,4)]
   Main> oddEvenPairs 7
   [(1,2),(1,4),(1,6),(3,2),(3,4),(3,6),(5,2),(5,4),
    (5,6),(7,2),(7,4),(7,6)]
   ```

1.2.5 Exercise 4: Using map and filter

List comprehensions can be replaced by using `map` and `filter`. Look at the type of these functions in the REPL.

1. Within the REPL, repeat the problem solved earlier using list comprehensions, but instead base your solution on `map`. Specifically, write an expression which uses `map` to produce the list of pairs (x, y) such that $y = 3x^2 + 2x + 1$ for $x \in 1, 2, \ldots, 10$.

   The result should be:

   ```
   [(1,6),(2,17),(3,34),(4,57),(5,86),(6,121),(7,162),
    (8,209),(9,262),(10,321)]
   ```

2. Repeat the previous exercise but use `map` and `filter` to return only those pairs whose second components are a multiple of 3.

These exercises show that list comprehensions are simpler and more readable alternatives.