Emma Perez

Haskell

I chose the language Haskell to do Homework 5 in. I thought that the transition from scheme to Haskell would be fairly smooth since both of them are functional languages. When I first started this project I tried to do the problems alone without looking at the scheme answers. Any of them that I didn't get I would look at the scheme answers and try to convert them to Haskell. The biggest issue I had with this was lambda, I wasn't exactly sure how to handle them.
Haskell is a functional language, statically typed, and is lazy.
It uses type definitions.
One picky and kind of annoying thing about Haskell is that it won't take tabs, only spaces. My text editor automatically adds tabs so I have to go back and delete hidden tabs.
I was also able to import certain modules to use for things like 'concat.'

```haskell
module Lib (module Data.List, module Data.Char, module Lib) where
import Data.List
import Data.Char


main :: IO ()
main = return()

1.
(yourname) = "Emma Perez"

2.
axb :: Float -> Float -> Float -> Float
axb a x b = ((a * x) + b)

3.
distance (x1, y1) (x2, y2) = sqrt (x * x + y * y)
    where
        x = x1 - x2
        y = y1 - y2

4.
purge match lst = filter (x) lst
    where
        not(x == match)
5.
counttrues lst = length (filter (==True) lst)

6.
buildlst a =
    if (a == 0)
    then []
    else (a : (buildlst (if (a < 0)
                        then ( a + 1)
                        else (a - 1))))

7.
dotproduct v1 v2 =
    if (length v1 == length v2)
    then (sum (zipWith (*) v1 v2))
    else sum v1

 8.
multiples base n =
    map (*base) (x)
    where
        x = buildlst n

9.
runcmdPlus x = sum(x)
runcmdTimes x = product(x)
runcmdCdr x = tail(x)
```

1. Exact same as Scheme

```
*Lib> (yourname)
"Emma Perez"
*Lib>
```

2. For problem #2, I had trouble defining “ax+b.” Haskell would not allow me to use a “+” In the definition of a function.

```
*Lib> axb 2 3 4
10.0
*Lib> axb 10 20 30
230.0
*Lib> axb 1 1 1
2.0
*Lib>
```

3.

```
*Lib> distance (1, 2) (3, 4)
2.8284271247461903
*Lib> distance (0, 0) (1, 1)
1.4142135623730951
*Lib>
```

4. This problem gave me issues when it came to filtering by the non matches. I kept getting this error.

```
[1 of 1] Compiling Lib              ( testing.hs, testing.o )

testing.hs:30:1: error:
    parse error (possibly incorrect indentation or mismatched brackets)
Emmas-MacBook-Air:SPL_Code emmaperez$
```

5. Wouldn’t let me add ‘-‘ to the name “count-trues” would give me parse error. When counting the occurrences of the Trues in a list. The filter function in Haskell needs two arguments, the first one being the conditional and the second being the list. The filter functions checks every item in the list against the condition and marks it True or False and then returns the trues. This function would not let me add random integers to the list because they would not return True or False. Had to put the list in brackets. Could not use ‘-‘ in “count-trues.”

```
*Lib> counttrues [False, False, True, True, True, True, 1 == 1]
5
*Lib> counttrues [True, True, True, True, True, True, False]
6
*Lib>
```

6. The only difference from scheme is that it needs parenthesis around the negatives.

```
*Lib> buildlst (-10)
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1]
*Lib> buildlst 10
[10,9,8,7,6,5,4,3,2,1]
*Lib> buildlst 0
[]
*Lib>
```

7. For the dotproduct, when trying to check to see if the length of each vector was equal I would get this error.

```
No instance for (Num Bool) arising from a use of ‘sum’
  • In the expression: (sum (zipWith (*) v1 v2))
    In the expression:
```

```
    if (length v1 == length v2) then
       (sum (zipWith (*) v1 v2))
    else
       False
  In an equation for ‘dotproduct’:
     dotproduct v1 v2
       = if (length v1 == length v2) then
            (sum (zipWith (*) v1 v2))
         else
            False
```

When I tried changing the ‘else’ from a Boolean to a Num, I got this error

```
testing.hs:40:10: error:
  • Occurs check: cannot construct the infinite type: t ~ [t]
  • In the expression: v1
    In the expression:
      if (length v1 == length v2) then (sum (zipWith (*) v1 v2)) else v1
    In an equation for ‘dotproduct’:
       dotproduct v1 v2
         = if (length v1 == length v2) then
              (sum (zipWith (*) v1 v2))
           else
              v1
  • Relevant bindings include
      v2 :: [t] (bound at testing.hs:37:15)
      v1 :: [t] (bound at testing.hs:37:12)
      dotproduct :: [t] -> [t] -> t (bound at testing.hs:37:1)
```

So I had to return another function.

```
*Lib> dotproduct [1,2,3,4] [1,2,3,4]
30
*Lib> dotproduct [1,2] [1,2,3,4,5]
3
*Lib>
```

8. Multiples was almost the same except for again the use of parenthesis when using negative numbers

```
*Lib> multiples 2 10
[20,18,16,14,12,10,8,6,4,2]
*Lib> multiples (-5) 12
[-60,-55,-50,-45,-40,-35,-30,-25,-20,-15,-10,-5]
*Lib> multiples 12 (-5)
[-60,-48,-36,-24,-12]
*Lib>
```

9. When attempting this problem, I ran into a series of issues. When I would create the cases for sum and product, the function worked fine. As soon as I would add concat (append) and tail (which does that same thing as cdr) I would get the error …

```
Occurs check: cannot construct the infinite type: a ~ [a]
```

On this code…

```
runcmd opname lst =
   if (opname == "plus")
   then sum lst
   else if (opname == "product")
   then product lst
```

```
else if (opname == "cdr")
then tail lst
else if (opname == "append")
then concat lst
else sum lst
```

I tried using case statements and regular if statements. Each function (plus, times, tail, and concat) work fine on their own but when added into cases they give errors.

```
• Occurs check: cannot construct the infinite type: a ~ [a]
    Expected type: [a]
      Actual type: [[a]]
  • In the expression: tail lst
    In the expression:
      if (opname == "cdr") then
          tail lst
      else
          if (opname == "append") then concat lst else plus lst
    In the expression:
      if (opname == "product") then
          product lst
      else
          if (opname == "cdr") then
              tail lst
          else
              if (opname == "append") then concat lst else plus lst
  • Relevant bindings include
      lst :: [[a]] (bound at testing.hs:52:15)
      runcmd :: [Char] -> [[a]] -> [a] (bound at testing.hs:52:1)
```

So I made each function separate …

```
runcmdPlus x = sum(x)
runcmdTimes x = product(x)
runcmdCdr x = tail(x)
runcmdAppend x = concat(x)
```

to get…

```
*Lib> runcmdPlus [1,2,3,4]
10
*Lib> runcmdTimes [1,2,3,4]
24
*Lib> runcmdCdr [1,2,3,4]
[2,3,4]
*Lib> runcmdAppend ["foo", "bar", " ", "Jones"]
"foobar Jones"
*Lib>
```

10. Since everything in Haskell has a type, it was really hard to make sure that my types and functions were compatible. That is the biggest issue I had with problem 10.