Homework 1 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Installing Haskell and GHCi

Done. Below is what I used to verify that I had completed the task:

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.6.1
$ ghci --version
The Glorious Glasgow Haskell Compilation System, version 9.6.1
```

> **NOTE:** Because I'm on Windows, I used GHCup to install GHC on my system. For reference, mine installed at version 9.2.7. I used `ghcup install 9.6.1 --set` to then install and set my version to 9.6.1, the latest version listed on the referred site on the spec at the time of this assignment.

Hello Haskell

```haskell
-- hello.hs
greeting = "Hello, world!"
```

```
$ ghci
GHCi, version 9.2.7: https://www.haskell.org/ghc/  :? for help
ghci> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.
ghci> length greeting
13
```

The length of the string is 13.

Installing Python 3

Done. Below is what I used to verify that I had completed the task:

```
$ which python3
/c/Users/vinlin/.pyenv/pyenv-win/shims/python3
$ python3 --version
Python 3.10.7
$ python3 -q
```

```
>>> print("Hello world!")
Hello world!
```

> **NOTE:** Python had already been installed on my (Windows) system prior to this assignment. Note that, similar to GHCup for GHC, I have pyenv-win installed to manage different Python versions, so the shell output above is not an error.

Haskell Warmup

Question 1

This question can be solved with a simple ternary expression, using a `where` clause to make it more readable.

```haskell
largest :: String -> String -> String
largest s1 s2 =
  if len1 >= len2
    then s1
    else s2
  where
    len1 = length s1
    len2 = length s2
```

```
ghci> largest "cat" "banana"
"banana"
ghci> largest "Carey" "rocks"
"Carey"
```

Question 2

This question is testing **operator precedence**. Barry encounters infinite recursion because a function call has higher precedence than arithmetic operators, so the snippet `reflect num+1` was being evaluated as if it was `(reflect num)+1` instead of `reflect (num+1)` like intended. This meant that the arguments to his recursive calls were never changing, so the algorithm never progresses and recurses forever.

```haskell
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect (num + 1)
  | num > 0 = 1 + reflect (num - 1)
```

```
ghci> reflect (-2)
-2
ghci> reflect 0
0
ghci> reflect 4
4
```

Question 3a

```haskell
all_factors :: Integer -> [Integer]
all_factors num =
  [x | x <- [1 .. num], (num `mod` x == 0)]
```

```
ghci> all_factors 1
[1]
ghci> all_factors 42
[1,2,3,6,7,14,21,42]
```

Coming from Python, this is simply the translation of the more readable:

```python
def all_factors(num: int) -> list[int]:
  return [x for x in range(1, num+1) if num % x == 0]
```

Question 3b

By definition, a perfect number is equal to the sum of its factors, excluding itself. Thus, we first get all the factors with our `all_factors` from above, sum all the elements, and then test if that's equal to `num` less than `num` as the comprehension guard.

```haskell
perfect_numbers =
  [num | num <- [1 ..], sum (all_factors num) - num == num]
```

```
ghci> take 4 perfect_numbers
[6,28,496,8128]
```

Question 4: Regular If Statements

This uses a technique called **mutual recursion**. We first define base cases for both functions using the assumption that all inputs are positive integers. Then, we can take advantage of the property that `num` and

`num-1` have opposite parities, so we can recursively call the other function with `num-1`.

```
is_odd :: Integer -> Bool
is_odd num =
  if num == 1
    then True
    else
      if num == 2
        then False
        else is_even (num - 1)

is_even :: Integer -> Bool
is_even num =
  if num == 1
    then False
    else
      if num == 2
        then True
        else is_odd (num - 1)
```

```
ghci> is_even 4
True
ghci> is_even 5
False
ghci> is_odd 4
False
ghci> is_odd 5
True
```

Question 4: Guards

Guards are a syntactic construct in Haskell that allow us to implement conditional branching. We specify a series of conditions, and the first one that is satisfied determines the control flow of the function. This allows us to use the same algorithm as above but in a more readable way:

```
is_odd :: Integer -> Bool
is_odd num
  | num == 1 = True
  | num == 2 = False
  | otherwise = is_even (num - 1)

is_even :: Integer -> Bool
is_even num
  | num == 1 = False
  | num == 2 = True
  | otherwise = is_odd (num - 1)
```

As a side note, `otherwise` is actually an alias for `True` defined in `Prelude`. This just serves to encourage more readable code.

I used the same tests as above to verify that I got it working.

Question 4: Pattern Matching

Pattern matching is another syntactic construct that allows us to implement conditional branching by matching specific patterns of values to call the corresponding binding of the function. Once again, we reuse the same algorithm:

```
is_odd :: Integer -> Bool
s_odd 1 = True
is_odd 2 = False
is_odd num = is_even (num - 1)

is_even :: Integer -> Bool
is_even 1 = False
is_even 2 = True
is_even num = is_odd (num - 1)
```

As a side note, it's a convention in wildcard matching to bind values to `_` if they do not need to be referenced in the body, similar to in Python.

I used the same tests as above to verify that I got it working.

Question 5

```
{-
The following code was generated by ChatGPT, with some modifications I made.

prompt: Write a Haskell function named count_occurrences that returns the number
of ways that all elements of list a1 appear in list a2 in the same order (though
a1's items need not necessarily be consecutive in a2). The empty sequence
appears in another sequence of length n in 1 way, even if n is 0. For example,
count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30] should return 3.

response (code part):

    count_occurrences :: Eq a => [a] -> [a] -> Int
    count_occurrences [] _ = 1
    count_occurrences _ [] = 0
    count_occurrences (x:xs) (y:ys)
      | x == y = count_occurrences xs ys + count_occurrences (x:xs) ys
      | otherwise = count_occurrences (x:xs) ys
-}
count_occurrences :: [Integer] -> [Integer] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (head1:a1) (head2:a2)
```

```
    | head1 == head2 = count_occurrences a1 a2 + count_occurrences (head1:a1) a2
    | otherwise = count_occurrences (head1:a1) a2
```

I looked to ChatGPT for this question as I was unable to solve it on my own. Based on the code snippet and the explanation it gave, I can paraphrase my own explanation on why this works. I realize from seeing the answer that it's important to be able to see these types of problems as MATH 61 problems, where I think of the problem in terms of itself - fundamentally, this is what recursion is about.

This implementation combines the techniques of pattern matching, guards, and recursion. We first use pattern matching to separate the obvious edge cases from the main recursive case. Based on the nature of the problem (and the test cases provided), we can immediately isolate the cases where either operand is an empty list.

- `[] _` should return 1 because there is one way to build `[]` as a sublist of any other list - namely the empty list, which is defined as a sublist of any list, including itself.
- `_ []` should return 0 because there is no way to build a sublist of the empty list unless `a1` is the empty list itself, but this is caught by the above case.

The main recursive case first uses pattern matching (with cons syntax) to deconstruct the list arguments into their heads and the rest of the lists. This is just a shortcut to using `head` and `tail` within the function body. We then use guards to separate the two cases that describe what we should do at this recursive step.

- If the current heads are equal, then we know we've found an element that COULD contribute to the final count. Specifically, we can choose whether to use this element as part of the sublist we're building or skip over it. For the part of the count where we include it, we advance both lists, so we recurse with the tail parts of the lists (`a1 a2`). For the part of the count where we skip over it, we still need to look for `head1` again, so we recurse with the original `a1` (`(head1:a1)`) but search through the remainder of `a2`.
- If they aren't, we simply search through the rest of `a2`, still looking for `head1`.

I verified that all the given test cases worked:

```
ghci> count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]
1
ghci> count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]
2
ghci> count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]
0
ghci> count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]
3
ghci> count_occurrences [] [10, 50, 40, 20, 50, 40, 30]
1
ghci> count_occurrences [] []
1
ghci> count_occurrences [5] []
0
```