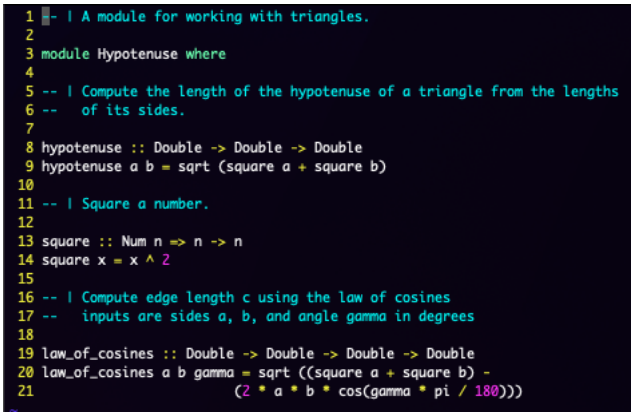
CMSC 22300 Problem Set 1

Will Thomas

January 9, 2023

Exercise 1.2



```haskell
-- | A module for working with triangles.

module Hypotenuse where

-- | Compute the length of the hypotenuse of a triangle from the lengths
--   of its sides.

hypotenuse :: Double -> Double -> Double
hypotenuse a b = sqrt (square a + square b)

-- | Square a number.

square :: Num n => n -> n
square x = x ^ 2

-- | Compute edge length c using the law of cosines
--   inputs are sides a, b, and angle gamma in degrees

law_of_cosines :: Double -> Double -> Double -> Double
law_of_cosines a b gamma = sqrt ((square a + square b) -
                      (2 * a * b * cos(gamma * pi / 180)))
```

Exercise 2.1

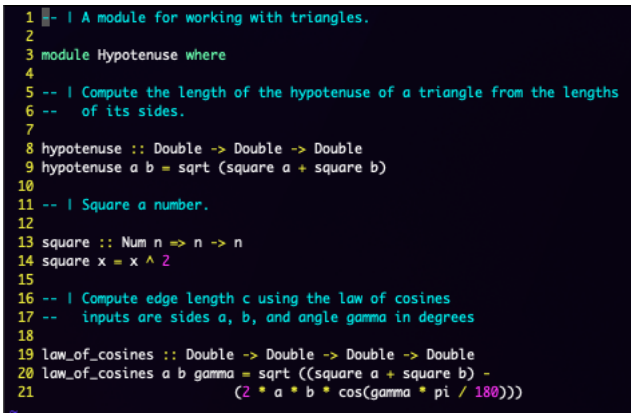
Consider

`sumf (sumf square) [[1,2],[3,4]]`.

Do a step-by-step substitution-based evaluation of this expression.

1. We begin with computing the outer sumf. We apply the function (sumf square) to each element in the list of lists [[1,2],[3,4]].
2. The outer sumf will isolate [1,2] via matching, at which point (sumf square) will use it as an input.
3. (sumf square) will, like the previous iteration of sumf, isolate 1 first via matching.
4. 1 will be squared and added to the not yet computed result from running sumf square on the rest of the list, which is just 2.
5. As such, when the base pattern is matched with, we achieve a total sum of 5.
6. Stepping back into the first iteration of sumf, we once again add 5 to the not yet computed result from computing (sumf square) again, but this time with [3,4] as the final input.
7. Stepping into (sumf square) once again, we know that 3 is isolated from the rest of the list. 3 is then squared and added to the not yet computed result of running sumf square on the rest of the list (just 4). Once again, when the base pattern is reached, we get a total of 25.
8. Stepping back into the first iteration of sumf, we cannot further process the list, so we reach the base pattern. Adding up the two parts, we reach a total of 30.

Exercise 2.2



```haskell
-- | A module for working with triangles.

module Hypotenuse where

-- | Compute the length of the hypotenuse of a triangle from the lengths
--   of its sides.

hypotenuse :: Double -> Double -> Double
hypotenuse a b = sqrt (square a + square b)

-- | Square a number.

square :: Num n => n -> n
square x = x ^ 2

-- | Compute edge length c using the law of cosines
--   inputs are sides a, b, and angle gamma in degrees

law_of_cosines :: Double -> Double -> Double -> Double
law_of_cosines a b gamma = sqrt ((square a + square b) -
                      (2 * a * b * cos(gamma * pi / 180)))
```

Exercise 2.3

NOTE: All methods evaluate to 90000.

```haskell
-- | A module showing three techniques of summing [0,...,99] under constraints

module First_100_sum where

-- | Given two integers d, n, return True if d evenly divides n
divisibleBy :: Integer -> Integer -> Bool
divisibleBy d n
    | n `mod` d == 0 = True
    | otherwise = False

-- | Given a list of predicates ps and a value x, return True iff p x is
--   True for every p in ps.

allp :: [(Integer -> Bool)] -> Integer -> Bool
allp [] x = False
allp (p:[]) x
    | p x = True
    | otherwise = False
allp (p:ps) x
    | p x = allp ps x
    | otherwise = False

-- | Combine filter with allp such that if a value fails the allp test
--   it is discarded

filterAll :: [(Integer -> Bool)] -> [Integer] -> [Integer]
filterAll ps (n:ns)
    | allp ps n = (n:(filterAll ps ns))
    | otherwise = filterAll ps ns
filterAll ps [] = []

result = sum
        . take 100
        . filter (divisibleBy 2)
        . filter (divisibleBy 3)
        . filter (not . divisibleBy 4)
        . filter (not . divisibleBy 9)
        $ [0..]

result2 = sum
        . take 100
        . filter (allp [divisibleBy 2,
                        divisibleBy 3,
                        not . divisibleBy 4,
                        not . divisibleBy 9])
        $ [0..]

result3 = sum
        . take 100
        . filterAll [divisibleBy 2,
                     divisibleBy 3,
                     not . divisibleBy 4,
                     not . divisibleBy 9]
        $ [0..]

```

Exercise 7.1

```haskell
-- | A sample module from the book to be modified

module Main where


import Data.Char
import System.Environment

capitalize :: String -> String
capitalize [] = []
capitalize (first:rest) = ((Data.Char.toUpper first) :rest)

main :: IO ()
main = do
    putStrLn "Hello. I am a HAL 9000 series computer."
    name <- getEnv "USER"
    putStrLn $ "Good morning, " ++ (capitalize name) ++ "."
```