FUNCTIONAL PROGRAMMING

Haskell

OPERATORS

Comments:

operator	description
--	start of comment for remainder of line
{- … -}	comment

```
main = do
-- Print a line
putStrLn "Greetings!  What is your name?"
inpStr <- getLine
putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

```
main = do
{-Print a line-}
putStrLn "Greetings!  What is your name?"
inpStr <- getLine
putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

OPERATORS (CONTINUED…)

Mathematical Operators:

operator	description
+	add
-	subtract infix minus - prefix
*	multiply
/	divide
^	raise to power
**	raise to power
&&	boolean and
||	boolean or
not	boolean not

OPERATORS (CONTINUED…)

Mathematical Comparison Operators:

operator	description
==	equal
/=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
mod	modulus

```
"ghci> " 46 `mod` 5
1

OR

"ghci> " mod 46 5
1
```

TAKE INPUT (FROM THE CONSOLE)

```
main = do putStrLn "What is 2 + 2?"
          x <- readLn
          if x == 4
              then putStrLn "You're right!"
              else putStrLn "You're wrong!"
```

```
doGuessing num = do
   putStrLn "Enter your guess:"
   guess <- getLine
   if (read guess) < num
     then do putStrLn "Too low!"
             doGuessing num
     else if (read guess) > num
            then do putStrLn "Too high!"
                    doGuessing num
            else putStrLn "You Win!"
```

SYNTAX FUNCTION (PATTERN MATCHING)

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

```
sayMe :: Int -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Output:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21):  Non-exhaustive
patterns in function charName
```

PATTERN MATCHING WITH TUPLES

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)
```

GUARDS

- We use *guards* when we want our function to check if some property of those passed values is true or false. That sounds a lot like an if expression, and it is very similar. However, guards are a lot more readable when you have several conditions, and they play nicely with patterns.

Let's dive in and write a function that uses guards. This function will berate you in different ways depending on your body mass index (BMI). Your BMI is calculated by dividing your weight (in kilograms) by your height (in meters) squared. If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25, you're considered normal. A BMI of 25 to 30 is overweight, and more than 30 is obese. (Note that this function won't actually calculate your BMI; it just takes it as an argument and then tells you off.)

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
 | weight / height ^ 2 <= 18.5 = "You're
underweight, you emo, you!"
 | weight / height ^ 2 <= 25.0 = "You're
supposedly normal. Pffft, I bet you're ugly!"
 | weight / height ^ 2 <= 30.0 = "You're fat!
Lose some weight, fatty!"
 | otherwise                   = "You're a whale,
congratulations!"
```

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I
 bet you're ugly!"
```

GUARDS (CONTINUED...)

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
 | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
 | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I
bet you're ugly!"
 | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
 | otherwise                   = "You're a whale, congratulations!"
```

Another way:

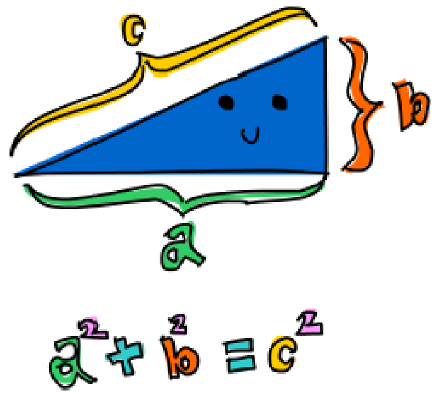
```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're
ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
   where bmi = weight / height ^ 2
```

RECURSION

Recursion is a function, which can call itself:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

PRACTICE



- Write a program in haskell that calculates the hypotenuse

(c) of the right triangle as shown in figure.

- Write a program with guards in haskell that can provide a grade with the following scale:

80-90 – A+, 70-79 – A, 60-69 – B, 50 – 59 – C and less than 50 is fail.

- If the end points of a line segment is (x_1, y_1) and (x_2, y_2) then write a program to find the midpoint of the line segment using tuples.

- Write a program in haskell that finds the Fibonnaci sequence until a given number using recursion.