RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC204A
Course Name	Advanced Programming Concepts
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	03/2018
Course Leader/s	V.S. Yerragudi

Declaration Sheet			
Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	03/2018
Course Code	CSC204A		
Course Title	Advanced Programming Concepts		
Course Date		to	
Course Leader	V.S. Yerragudi		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

Table No.	Title of the table	Pg.No.

Figure No.	Title of the figure	Pg.No.

Question No. 1

Solution to Question No. 1 Part A:

A 1.1 Introduction with relevance of the debate:

The topic of the debate is "Functional Programming must be used to solve all programming problems due to its expressivity." Functional programming has been long popular in academia, but with a few industrial applications However in the recent years several prominent functional programming languages have been used in commercial or industrial systems. An example being Erlang which was developed by a Swedish company Ericsson in the late 1980s, has become popular for building a range of applications at companies such as Facebook and WhatsApp.

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

Functional Programming is based on Lambda Calculus, Lambda calculus is framework developed by Alonzo Church to study computations with functions. It can be called as the smallest programming language of the world. It gives the definition of what is computable. Anything that can be computed by lambda calculus is computable. It is equivalent to Turing machine in its ability to compute. It provides a theoretical framework for describing functions and their evaluation. It forms the basis of almost all current functional programming languages.

A 1.2 Stance taken with Justification:

The First Paper "Why functional Programming Matters" by J. Hughes, tries to convince the 'real world' that functional programming is vitally important and to help the functional programmers exploit the advantages of functional programming by clarify those advantages.

So, what is a functional programming language? Here's what the paper says, "Functional programming is so called because a program consists entirely of functions. The main program itself is written as function which receives the program's input as its argument and delivers the program's output as its result", "Functional programs contain no assignment statements", "there are no side effects that change the value of an expression". And then goes on explaining all the advantages functional programming has over imperative counterparts using these special features exclusive to functional programming.

The Second Paper “Why no one uses functional languages” by Philip Wadler, advocates precisely what the title says, its states facts and reasons of why functional languages aren’t as popular as the imperative ones like C, C++, and Java.

I do not agree with the statement stated in the Introduction, that Functional programming must be used to solve all programming problems due to its expressivity. Both the papers are right on their parts and do a great job in explaining their purpose. Albeit as of the topic of the debate, choosing a programming language not only depends upon its expressivity but also its paradigm supported, domain of application, scale of programming, Popularity, Portability, Training, Performance and Future Support.

Paradigm and Training is also vital when choosing the language, not everyone can grasp the functional realm, programmers practiced in imperative languages are used to a certain style of programming. For a given task, the imperative solution may leap immediately to mind or be found in a handy textbook, while a comparable functional solution may require considerable effort to find (even if once found it is more elegant).

Coming to the Portability, Testing and Performance, taking the example of Haskell, that is a real pain when installing the different package managers on different platforms, and so called an “adventure” of process. Constructing debuggers and profilers for lazy languages such as Haskell is recognized as difficult, and hence building an IDE for it is difficult, and consist of lot of work and research. Such tools are sorely needed for an Industrial Level development. Quoting the paper: “I have heard of numerous projects where C won out over a functional language, not because C runs faster (although often it does), but because the hegemony of C guarantees that it is widely portable.” Coming to Performance, since many of the functional languages are built on top of C/C++ they are relatively fast but still the performance is significantly inferior to C. This is because our systems/machines are constructed imperatively, they behave imperatively, there are states, and objects have behavior, the state is changed, and functional paradigm seems more like an emulation on such machine.

A 1.3 Conclusions drawn from the paper:

Concluding, there are a large number of factors along with expressivity that are important when choosing a programming language, sure Functional Programming is really good at what it is, but not every problem can be expressed functionally in an efficient manner, some imperative solution will always be superior compared to its functional counterpart, that’s inevitable, the domain of the problem plays a major part in choosing the language. “Despite the applications work listed above, functional programming researchers place far more emphasis on developing systems than on applying those systems. Further, the bulk of effort is devoted to language design, program analysis, and the construction of optimizing compilers, with far less to debuggers, profilers, and software engineering tools and methodologies” as stated by Wadler.

Question No. 2

Solution to Question No. 1 Part B:

B 1.1 Introduction to problem:

The given question is a mathematical one, that demands for a program that should be able to perform tasks which are mathematical in nature, the program specifications include:

1. Generate the nth prime
2. Test if given number is prime
3. Test if two numbers are relatively prime

Solutions to the above specification can be expressed mathematically in terms of functions and sub functions, Generating the nth prime has a non-deterministic polynomial time of execution, i.e. the time taken to find the nth prime is non-deterministic, although since the solution still can be expressed in a mathematical form, it should be representable in a Programming Language, the choice of the language will matter, since not all programming language will support a lazy evaluation, and we need a lazy evaluation, this is further discussed in the later part of the assignment.

The other two specifications are pretty straight forward, as there are concrete mathematical definitions for them, although choosing the right algorithm would be another task for them.

We define a prime number as a number that has only 2 divisors, the number itself and 1. In that sense 0 has infinite divisors, 1 has only 1. This logic will be used to generate the nth prime and check for a prime, the relatively prime or coprime, in the mathematical domain is strictly defined as the two numbers whose gcd is 1.

B 1.2 UI Selection:

Since the problem works on a mathematical realm, the given program specifications can be represented as mathematical functions, and the programming paradigm that is close to such kind of thing is Functional Programming, where each of the mathematical function can be written as they are or close to they are in a Functional Programming Language, one such language is Haskell, which we would use for such a scenario.

Functional Programming language makes it easy to express our solution since we can define the solution in terms of mathematical functions, these functions are made of more sub functions, hence they can be decomposed functionally. Another benefit of Functional is that there are no side-effects, so it’s not prone state problems encountered in paradigms such as imperative one. Recursion is a very powerful technique that can be used in this to express the solution, we will see later how recursion is used here.

Haskell is chosen instead of any other language is because Haskell is a pure functional language and moreover the Haskell code using ghc compiles to C++ code which is executed on the machine, this makes it faster than any other alternative out there. Moreover, the Haskell documentation is way better than other functional languages so it's easy to find solutions when stuck at a problem.

The program needs to be simple to use by a mathematician and hence GUI is not necessary, this can be done using a CLI or Command Line Interface. This is Action-Object Interface, in which the action to be performed, which will either be to test for prime, or to generate the nth prime, or to check for coprime, is provided before the objects are given, i.e. the arguments for that action. The CLI Interface is very easy to use and very intuitive, Mathematicians are used to Command Line Interfaces such as MATLAB or Octave. This will make it easier to use for them, since they are already aware of how such interfaces work.

The main motto being KIS -> Keep It Simple, and most importantly Elegant Code and an Intuitive Interface to use is important.

B 1.3 Functional Decomposition:

The following that the program specifications broken down into their sub functions.

1. `Primes` is defined as:

$$\text{Primes} = \{x \in \mathbb{N} - \{1\} \mid \text{P}(x)\}$$
$$\text{where } P(x) = \text{isPrime}(x)$$

$$\text{Primes} : x \subset \mathbb{N}$$

<u>Discussion:</u>
We already know the fact that primes are always greater than 1, i.e. the smallest prime is 2, and every primes is also a natural number, hence we define our range to be a subset of Natural Numbers.
Using the Set Builder form the Set of Primes can be represented as every element of x, which is a Natural Number such that x is a Prime. Which in predicate logic can be written as $\text{P}(x) = x$ is a prime.

<u>Summary:</u>
Name: `Primes`
Parameters : ϕ
Return Type : $\{x \mid x \subset \mathbb{N}\}$
Description : `Returns the list of Primes`
Where $\phi = \{\}$

2. `isPrime` is defined as:

$$\text{isPrime}(x) = \forall p \in \{\, y \ \in \text{Primes} \mid p^2 < \ y\}\,, \text{M}(x,p)$$
$$\text{where M}(x,p) = x \bmod p > 0$$

$$\text{isPrime} : \ \mathbb{N} \rightarrow \text{Bool}$$
$$\text{where Bool} = \{\text{True}, \text{False}\}$$

Discussion:

`isPrime` is a function that would check if the given number is a prime or not, This Function is a mapping from a Natural Number to a Boolean, where Boolean is True or False.

The logic behind this is the Sieve of Eratosthenes, we know for the fact that for a number to be prime, it has to have only 2 factors, which is 1 and the number itself, also that means if any element from 2 to (number-1) divides our number completely (mod is 0) then our number is not prime. All sieve does not use every number from 2 to the number – 1, instead we use only those numbers from the set of primes that are less than $\sqrt{x}$ $(p < \sqrt{x}\ or\ p^2 < x)$, this optimizes our code and time complexity. So, in short if all numbers from the list of primes that are less that square root x, remainder with x give greater than 0 value then the number x is prime.

Summary:

Name: `isPrime`

Parameters : $\mathbb{N}$

Return Type : $\{x \mid x \ \in \{True \ \ , False\}\}$

Description : `Returns if the given number is a prime or not`

3. `coprimes` is defined as:

$$\text{coprime}(x,y) = \text{gcd}(x,y) = 1$$
$$\text{where gcd}(x,y) = \begin{cases} \quad x, \quad y = 0 \\ \text{otherwise} = \text{gcd}(y, x \bmod b) \end{cases}$$

$$\text{coprime} : \mathbb{N}^2 \rightarrow \text{Bool}$$
$$\text{where Bool} = \{\text{True}, \text{False}\}$$

$$\text{gcd} : \ \mathbb{N}^2 \rightarrow \ \mathbb{N}$$

Discussion:

Coprime is simply defined as those two numbers who have a gcd of 1, gcd uses the Euclid's algorithm to compute the gcd of two numbers.

The function Coprime is a mapping from $\mathbb{N}^2$ to Bool, which is a point on a 2D Space, where the coordinate is a natural number to a Boolean value, and GCD is a mapping from $\mathbb{N}^2$ to $\mathbb{N}$, i.e. a point on a 2D space, where the coordinates is a natural number to a point in 1D space where the point coordinate is a natural number. In simple terms coprimes maps from two natural numbers to a boolean and gcd maps from two natural numbers to a natural number.

Summary:

Name: coprime

Parameters : $\mathbb{N}^2$

Return Type : $\{x \mid x \in \{True\ ,False\}\}$

Description : `Returns if the two numbers are coprime`

Name: gcd

Parameters : $\mathbb{N}^2$

Return Type : $\mathbb{N}$

Description : `Returns the gcd of the two numbers`

B 1.4 Conclusion

In Conclusion we can see that its very easy to express the program specifications mathematically and perform a functional decomposition to get the sub functions, these sub functions can be used independently and also with other functions. Expression the solution in Mathematical form makes it easier for the implementation during the development phase, since the solution in code is almost similar or same as that of the solution expressed here.

The functions written here conform to the program specifications and output the required result, which was verified to be mathematically true.

As we already discussed Action-Object Interface is chosen as the Interface, since the output is basically just numbers and truth values, a GUI is unnecessary.

Choosing all these parameters for the program beforehand the development makes gives a clear picture of how the program skeletal looks like and makes it easier while implementation.

Solution to Question No. 2 Part B:

B 2.1 Introduction to development:

Since we are working with mathematical functions, that are to be implemented in a functional programming language, we first write the mathematical functions in a normal mathematical way, and then express them in a Functional Programming way. Since there are some restrictions that come into picture when trying to implement in a Programming Language, these are discussed in the discussions in B2.2.

This first step would be to look for methods to implement the Command Line interface in Haskell, which happens to be supported in Haskell, the next step is to find the equivalent inbuilt functions and Data types that can help in the development process of the program. This would decrease the development time and increase the programmer's efficiency.

After the equivalent functions and data types in Haskell are found, the program is then implemented from the Mathematical functions that were formed in the earlier part of this assignment.

Since Haskell is an interpreted language, during the development the decomposed functions can be checked if they are working as they should simultaneously, there's no need to compile, this saves time during the development and testing phase.

Also, the Prime Number functions is separate from the Main.hs module and hence a separate Library module is created for it, this is then imported in Main.hs and the PrimeFunctions can be used, also the Print Utility should be separated from the Main.hs and another Library is created from that. All these are then imported in the Main.hs file. This is following the Haskell standards, for future scope, suppose someone wants to use the Prime Numbers library, they can easily import this Library and use in their program as they desire.

B 2.2 Implementation of methodology:

1. `primes`

```haskell
primes :: [Integer]
primes = 2 : [ x | x <- [3, 5..], isPrime primes x]
```

Summary:

Name: `Primes`

Parameters : `None`

Return Type : `[Integer]`

Description : `Returns the list(Integer) of Primes`

Discussion:

primes is a list primes, which is a lazy list, i.e. the rest of the items in the list are not known until it is evaluated, We know that the primes start from 2, hence the first item in the list is 2, and every element in the list must be a prime, this function to check if it is a prime is defined later, but this is where the function is decomposed, also we know for the fact that all primes except 2 are odd, so we only want to keep odd numbers, hence we use 3, 5, . ., in Haskell this will do pattern matching is keep only odd numbers in the list.

The cons operator appends 2 to the list, and every element in the list [3, 5..] are checked with with predicate isPrime, if is true then its added to the list, and the list grows on and on.

There is no parameter to this function, and this returns a list of primes, the elements in the list are of Integer types, since we do not define a range of the digits on a prime number, this makes it easy to express the solution closer to the mathematical function we defined earlier in part B1.

2. `isPrime`

```haskell
isPrime :: [Integer] -> Integer -> Bool
isPrime _ 0 = False
isPrime _ 1 = False
isPrime primesList x
    | x < 0 = throw NegativeNumberException
    | otherwise = all (\p -> x `mod` p > 0) (trialFactors x)
        where
            trialFactors x = takeWhile (\p -> p * p <= x) primesList
```

Summary:

Name: `isPrime`

Parameters : `Integer`

Return Type : `Bool`

Description : `Returns if the given number is a prime or not`

Discussion:

The first thing when implementing is to take care that our function must only work for natural numbers since we defined our mathematical function in that way, so we have to exclude all the negative number, we check if the number is negative and throw a `NegativeNumberException` if the number is found to be negative, this exception can be caught in the driver function and necessary actions are to be taken.

To improve the code efficiency, we have defined two kinds of list of prime numbers, one function that takes up more space but the time complexity is improved and the other where it takes less space but the time complexity is bad. The implementation of improved sieve can be found in the Appendix A of this assignment.

By usage of Pattern-Matching in Haskell we declare that 0, 1 are not Primes and we have already checked for negative numbers, so the otherwise condition takes care of the positive Natural Numbers except 1.

Let's see how the function works by taking an example

```
>> isPrime primes 11
```

Here we have passed 11 as the argument to check for primes along with the list of primes. We'll assume that the list of primes is already generated, which is fair enough, because the list is lazy evaluated in Haskell.
The number is neither negative, not its 0 or 1, so the otherwise condition is executed,

Lets first make up for the trial factors, which are numbers that are $< \sqrt{11}$, i.e. < 3.31, the primes that are less than that are $2, 3$. We could have used $sqrt$ as the mathematical expression, but in Haskell this will convert the Integer into a fraction and then causing more problems, hence we square both sides to get $p^2 \leq x$, this way we avoid the Data Type Conversion and the mathematical meaning also does not change.

The equivalent for ∀ (for all) of Maths is all in Haskell. takeWhile keeps taking the items of the list until the predicate/lambda function in our case return a False.

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

all is a function that takes in a predicate or a lamda function that returns a Bool and folds that to a list of items and returns a Bool, basically it sees if that lambda function is True for all the items in that list.
The lambda function is $(\backslash \mathrm{p} \rightarrow \mathrm{x} \,\grave{}\mathrm{mod}\grave{}\, \mathrm{p} > 0)$, so we perform this to the list we obtained before.

$$11 \,\grave{}mod\grave{}\, 2 > 0 : True$$

$$11 \,\grave{}mod\grave{}\, 3 > 0 : True$$

Since for all the items in the list it is true, then the number passed as the argument i.e. 11, must be a prime, True, which is returned.

3. `coprime`

```haskell
coprime :: Integral a => a -> a -> Bool
coprime x y = gcd' x y == 1
```

Summary:

Name: coprime

Parameters : `Integer, Integer`

Return Type : `Bool`

Description : `Returns if the two numbers are coprime`

Discussion:

coprime is also pretty straight forward, we take in any Integral value, two of them, and return a Boolean, which is Bool in Haskell, that can be either True or False, going by the definition of coprime, two number are coprime or relatively prime if their gcd is 1, hence we perform functional decomposition again here and just write in that if the gcd x y == 1 then the numbers are coprime, the definition of gcd is defined later.

4. gcd

```haskell
gcd' :: (Integral a) => a -> a -> a
gcd' x y
    | y == 0 = x
    | otherwise = gcd' y (x `mod` y)
```

Summary:

Name: gcd

Parameters : `Integer, Integer`

Return Type : `Integer`

Description : `Returns the gcd of the two numbers`

Discussion:

The implementation of gcd is exactly similar to that to that of our mathematical expression, we take two Integral numbers as the argument and return an Integral value. This is why Haskell is was chosen to express the solution, it makes it very easy to write code which is already expressed in a mathematical form.

gcd uses the Euclidean algorithm to find the gcd, which is a pretty fast algorithm

`main Function, or the Driver Function`

```haskell
main :: IO ()
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    (action args) `catches` catchIt
```

Discussion:

The main method does not take any arguments but returns a IO(), the arguments are taken from the command line and passed to `(command:args)` using the `getArgs`

```
getArgs :: IO [String]  -- Defined in `System.Environment'
```

the arguments are then looked up in a function dispatch table (see appendix A for complete implementation), and the pattern that matches is then called function and the args is sent to that function.

If any Exception occurs while the execution of the function, such as the NegativeNumberException, or any other unknown exception, is then caught by `catchIt` (see appendix A for complete implementation).

B 2.3 Comment on the Results:

See Appendix B : Output and Testing for complete Testing Output

The Output obtained conforms to the Program specifications and performs the required operations, it also has exception handling and the program is tested for various kinds of inputs.

Output:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -p 23
[(23,True)]
```

Discussion:

The executable code is run with arguments -p 23, here -p is interpreted as the command and “23” is interpreted as the argument, this is passed to the main function as (command:args) which performs a lookup from the list of dispatch functions, since -p matches something in that table, it then calls isprime method and passes [“23”] which is the list of arguments to the isprime function, isprime function then parses the list of arguments and maps each of the items in the list which are strings into numbers, which we want, this is done using the map function

```
map :: (a -> b) -> [a] -> [b]    -- Defined in `GHC.Base'
```

This function takes a function as an argument, which can map an item to another item, map then executes this function on every item in the list, therefore transforming the list, and returns the transformed list. After we obtain the list of numbers, we use map function again to map isPrime function imported from PrimesLib to every item on the numbers list.

Any Exception that happens such as negative number, is thrown from the isPrime function in the PrimesLib and caught by catchIt in Main.hs. The final list is then printed in the command line output using the helper method pprint, which stands for pretty print, defined in PrintUtilLib.hs, what it basically does is that it prints the input that the user gave along with the output obtained, this makes it easier to read the output.

Output:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -np 21
```

```
[("21",73)]
```

Discussion:

Once again here, -np is taken as the command and “21” as the argument, the command “-np” is looked up in the dispatch table and nthprime function is executed, which is passed with the arguments list [“21”], then then converts this list of numbers in strings, into numbers using `read :: Int` function. And then it executes nthPrime function from PrimesLib.hs to every number in the list, the output obtained is passed to pprint along with the input to pretty print the result.

B 2.4 Conclusion:

Development of the program is very simple and easier to express in Haskell, which is a Functional Programming Language and also because the chosen Programming Language is very close the Solution of the Mathematical problem. The Solution in Haskell is almost similar or exactly same to that of the proposed mathematical functions that solves the problem which was discussed earlier in B1.

The documentation of Haskell is on-point and provides accurate information for it’s supported functions, also since it’s a popular functional programming language most of the questions are answered on stackoverflow, which is a major help during development.

The program can be further improved to support parallel computing, since the processors have multi-cores and multi-threads, just using one or two of them is not efficient, all of the cores can be used. This was the efficiency of the program becomes much more than single threaded one. Better algorithms can be implemented since not all algorithms have the same time and space complexity, one may want to use a algorithm that takes more space and less time since he has a machine with more RAM, and there might be someone who has less RAM and they could use the algorithm with higher time complexity but less space complexity.

1. For complete Source and Build Instructions : https://github.com/satyajitghana/University-Work.
2. https://www.geeksforgeeks.org/functional-programming-paradigm/ .
3.

PrimesLib.hs

```haskell
module PrimesLib ( primes, isPrime, kprimes, coprime, gcd , PrimeError) where

import Control.Exception
import Data.Typeable

data PrimeError = NegativeNumberException | MiscError String

instance Show PrimeError where
    show NegativeNumberException = "Prime Numbers is not defined for Negative Numbers"
    show (MiscError str) = "Unexpected " ++ str

instance Exception PrimeError

-- generates a list of prime numbers
primes :: [Integer]
primes = 2 : [ x | x <- [3, 5..], isPrime primes x]

-- checks if a given number is prime, takes the list of primes and the number to be
checked for
-- prime as input, returns Bool
isPrime :: [Integer] -> Integer -> Bool
isPrime _ 0 = False
isPrime _ 1 = False
isPrime primesList x
    | x < 0 = throw NegativeNumberException
    | otherwise = all (\p -> x `mod` p > 0) (trialFactors x)
        where
            trialFactors x = takeWhile (\p -> p * p <= x) primesList

-- A relatively simple but still quite fast implementation of list of primes. By Will
Ness
-- http://www.haskell.org/pipermail/haskell-cafe/2009-November/068441.html
-- generates a list of primes numbers using an optimized sieves method
kprimes :: [Integer]
kprimes  = 2: 3: sieve 0 primes' 5  where
    primes' = tail kprimes
    sieve k (p:ps) x
        = noDivs k h ++ sieve (k+1) ps (t+2)
        where (h,t) = ([x,x+2..t-2], p*p)
    noDivs k = filter (\x-> all ((/=0).(x`rem`)) (take k primes'))

-- checks if two given numbers are co-prime or not
coprime :: Integral a => a -> a -> Bool
coprime x y = gcd' x y == 1

-- computes the gcd of two given numbers
gcd' :: (Integral a) => a -> a -> a
-- gcd' a 0 = a
-- gcd' a b = gcd' b (a `mod` b)
gcd' x y
    | y == 0 = x
    | otherwise = gcd' y (x `mod` y)
```

PrintUtilLib.hs

```haskell
module PrintUtilLib ( pprint ) where

-- a print utility for pretty printing the primes output
pprint :: [a1] -> [a2] -> (a2 -> b) -> [(a1, b)]
pprint first second func = zip first (map func second)
```

Main.hs

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
module Main where

import System.Environment
import Control.Exception
import Data.Maybe
import PrimesLib
import PrintUtilLib

-- reads a String as an Integer
readNum :: String -> Integer
readNum p = read p :: Integer

-- isprime : takes in the numbers which are to be checked for prime
isprime :: [Integer] -> [String] -> IO ()
isprime primesList nums
    | length nums <= 0 = error "Enter at least one number"
    | otherwise = print (pprint numbers numbers (isPrime primesList))
        where numbers = map readNum nums

-- nthprime : prints the nth prime from the list of primes
nthprime :: [Integer] -> [String] -> IO ()
nthprime primesList nums
    | length nums <= 0 = error "Enter at least one number"
    -- gives just the list of the nth primes
    -- | otherwise = print ( map ((\ q -> primes !! (q - 1)) . (\p -> (fromIntegral p) :: 
Int ) . readNum) nums )
    -- zips the input and the output for prettier and understandable output
    | otherwise = print (pprint nums numbers nprime)
        where   numbers = map ( (\p -> (fromIntegral p) :: Int) . readNum) nums
                nprime = (\q -> primesList !! (q - 1))

-- arecoprime : checks if two given numbers are coprime
arecoprime :: [String] -> IO ()
arecoprime nums = do
    let mynums = ((map (\p -> read p :: Integer)  nums))
    print (coprime (mynums!!0) (mynums !!1))


dispatch :: [(String, [String] -> IO ())]
dispatch =  [("-isPrime", (isprime primes))
            ,("-p", (isprime primes))
            ,("-kprime", (isprime kprimes))
            ,("-kp", (isprime kprimes))
            ,("-nthPrime", (nthprime primes))
            ,("-np", (nthprime primes))
            ,("-knthPrime", (nthprime kprimes))
            ,("-knp", (nthprime kprimes))
            ,("-areCoprime", arecoprime)
            ,("-cp", arecoprime)
            ,("-help", help)
            ,("-h", help)]

-- prints the help for different commands, if args is empty, prints the program USAGE
```

```haskell
help :: [String] -> IO()
help x
    | length x <= 0 = do
        file <- readFile "./USAGE.txt"
        putStrLn file
    | otherwise = do
        file <- readFile "./HELP.txt"
        print ( map (fromJust.(\p -> lookup p (cmdHelp' file))) x )

-- reads the HELP.txt and returns a list of tuple with the help content
cmdHelp' :: String -> [(String, String)]
cmdHelp' file = map (\line -> read line :: (String, String)) (lines file)

-- list of the handles and the repective exception
catchIt :: [Handler ()]
catchIt = [Handler (\(err :: PrimeError) -> putStrLn $ "PrimeError: " ++ show err),
           Handler (\(err :: PatternMatchFail) -> putStrLn $ "Parsing Error, option not 
found, try \"-help\""),
           Handler (\(err :: SomeException) -> putStrLn $ "Unexpected Error !, try \"-
help\" for program USAGE\n")]

main :: IO ()
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    (action args) `catches` catchIt
```

USAGE.txt

```
Program : PrimeNumberFunctions
Options :
	"-isPrime" : Computes if the given number is a prime, can take a list of values as 
input
	USAGE : ./PrimeNumberFunctions -p 1 2 3 4 5
	ShortHand: "-p"

	"-kprime" : Computes if the given number is a prime, can take a list of values as 
input, uses kprime algorithm
	USAGE : ./PrimeNumberFunctions -kp 1 2 3 4 5
	ShortHand: "-kp"

	"-nthPrime" : Computes the nth Prime, can take a list of values as input
	USAGE : ./PrimeNumberFunctions -np 17 6 1999
	ShortHand: "-np"

	"-knthPrime" : Computes the nth Prime, can take a list of values as input, uses 
kprime algorithm
	USAGE : ./PrimeNumberFunctions -knp 17 6 1999
	ShortHand: "-np"

	"-areCoprime" : Checks if the given two numbers are coprime
	USAGE : ./PrimeNumberFunctions -cp 2 3
	ShortHand: "-cp"

	"-help" : Prints the documentation for the prime function
	USAGE : ./PrimeNumberFunctions -h isPrime
	ShortHand: "-h"

MIT License

Copyright (c) 2018 Satyajit Ghana
```

HELP.txt

```
("isprime", "checks if each of the numbers passed to it are prime or not, outputs Bool")
("nthprime", "computes the nth prime")
```

OUTPUT and Testing:

Help Function:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -h

Program : PrimeNumberFunctions
Options :
        "-isPrime" : Computes if the given number is a prime, can take a list of
values as input
        USAGE : ./PrimeNumberFunctions -p 1 2 3 4 5
        ShortHand: "-p"

        "-kprime" : Computes if the given number is a prime, can take a list of
values as input, uses kprime algorithm
        USAGE : ./PrimeNumberFunctions -kp 1 2 3 4 5
        ShortHand: "-kp"

        "-nthPrime" : Computes the nth Prime, can take a list of values as input
        USAGE : ./PrimeNumberFunctions -np 17 6 1999
        ShortHand: "-np"

        "-knthPrime" : Computes the nth Prime, can take a list of values as
input, uses kprime algorithm
        USAGE : ./PrimeNumberFunctions -knp 17 6 1999
        ShortHand: "-np"

        "-areCoprime" : Checks if the given two numbers are coprime
        USAGE : ./PrimeNumberFunctions -cp 2 3
        ShortHand: "-cp"

        "-help" : Prints the documentation for the prime function
        USAGE : ./PrimeNumberFunctions -h isPrime
        ShortHand: "-h"

MIT License

Copyright (c) 2018 Satyajit Ghana
```

Help for a specific function:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -h isprime

["checks if each of the numbers passed to it are prime or not, outputs Bool"]
```

Testing for Prime

Checking for a list of primes:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -p 123 1237327 1276761237 19219 1 0 17 23 19319

[(123,False),(1237327,False),(1276761237,False),(19219,True),(1,False),(0,False)
,(17,True),(23,True),(19319,True)]
```

Checking for really large numbers:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -kp
99991273618273681263876127863781627836781268736781263786129129391232131132123124
2133243247238461

[(99991273618273681263876127863781627836781268736781263786129129391232131132123l
24213324324723846l,False)]
```

Checking for negative numbers:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -p -12

PrimeError: Prime Numbers is not defined for Negative Numbers
```

Generating the nth Prime

Generating when n is given:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -np 100000

[("100000",1299709)]
```

Generating nth prime when the list of n is given:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -np 1 10 100 1000 1000

[("1",2),("10",29),("100",541),("1000",7919),("1000",7919)]
```

For negative n:

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -np -10

Unexpected Error !, try "-help" for program USAGE
```

Checking coprimes

```
PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -cp 123 246

False

PS D:\Assignment-SEM03-02-2018\CSC204A\PrimeNumberFunctions>
.\PrimeNumberFunctions.exe -cp 1231 7

True
```