1. Give Haskell expressions, with associated types to

(a) multiply two Double precision numbers, 3.1416 and 4.7
3.1416 * 4.7

(b) concatenate “Hello” and “World”
"Hello" ++ "World"

(c) list the integers from -1000 to 24000
[-1000 .. 24000]

2. (a) Write a Haskell function \texttt{sumTill :: Int -> Int}, so that \texttt{sumTill n} calculates the sum of the numbers between 1 and (positive) \texttt{n},

\begin{verbatim}
sumTill :: Int -> Int
sumTill 0 = 0 -- Works for n > 0
sumTill n = n + sumTill (n-1)
\end{verbatim}

(b) Show how \texttt{sumTill 3} reduces to 6

\begin{verbatim}
sumTill 3
=> 3 + sumTill 2
=> 3 + 2 + sumTill 1
=> 3 + 2 + 1 + sumTill 0
=> 3 + 2 + 1 + 0
=> 6
\end{verbatim}

3. Write a Haskell function \texttt{fib :: Int -> Int}, so that \texttt{fib n} calculates the fibonacci number for \texttt{n}.

\begin{verbatim}
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
\end{verbatim}

4. (a) Write a Haskell function \texttt{prod :: [Int] -> Int}, that returns the product of the elements of the list argument.

\begin{verbatim}
prod :: [Int] -> Int
prod [] = 1
prod (x:xs) = x * prod xs
\end{verbatim}

(b) Show how \texttt{prod [3 .. 5]} reduces to 60.

\begin{verbatim}
prod [3 .. 5]
=> prod [3,4,5]
=> 3 * prod [4,5]
=> 3 * 4 * prod [5]
=> 3 * 4 * 5 * prod []
=> 3 * 4 * 5 * 1
=> 60
\end{verbatim}
(c) Use a higher order function to write \texttt{prodH}, with the same semantics as \texttt{prod}.
\[
\texttt{prodH} :: \texttt{[Int]} \rightarrow \texttt{Int}
\texttt{prodH} \texttt{xs} = \texttt{foldr} (\times) 1 \texttt{xs}
\]

(d) Write a function \texttt{prodG} (for Generic) to calculate products of lists of any numeric type.
\[
\texttt{prodG} :: \texttt{Num a} \Rightarrow \texttt{[a]} \rightarrow \texttt{a}
\texttt{prodG} \texttt{xs} = \texttt{foldr} (\times) 1 \texttt{xs}
\]

5. Write a Haskell function \texttt{positive} :: \texttt{Int} \rightarrow \texttt{Bool}, so that \texttt{positive \ n} is true if \(n\) is greater than 0, and false otherwise.
\[
\texttt{positive} :: \texttt{Int} \rightarrow \texttt{Bool}
\texttt{positive \ n}
\mid \texttt{n > 0} \quad = \texttt{True}
\mid \texttt{otherwise} \quad = \texttt{False}
\]

6. Write a Haskell function \texttt{intersect} :: \texttt{Eq a} \Rightarrow \texttt{[a]} \rightarrow \texttt{[a]} \rightarrow \texttt{[a]}, so that \texttt{intersect \ xs \ ys} returns a list containing only those elements of \texttt{xs} that also appear in \texttt{ys}.
\[
\texttt{intersect} :: \texttt{Eq a} \Rightarrow \texttt{[a]} \rightarrow \texttt{[a]} \rightarrow \texttt{[a]}
\texttt{intersect \ xs \ ys} = [ x \mid x \leftarrow \texttt{xs}, x \ 'elem' \ \texttt{ys}]
\]

7. (a) Write a Haskell function \texttt{powers} :: \texttt{Int} \rightarrow \texttt{[Int]}, so that \texttt{powers \ n} returns an infinite list \([n^1, n^2, n^3, ...]\)
\[
\texttt{powers} :: \texttt{Int} \rightarrow \texttt{[Int]}
\texttt{powers \ n} = [ n^x \mid x \leftarrow [0..] ]
\]
(b) Write a Haskell expression to return the first 5 powers of 3
\[
\texttt{take 5 (powers 3)}
\]

8. Write a Haskell highest common factor function: \texttt{hcf} :: \texttt{Integer} \rightarrow \texttt{Integer} \rightarrow \texttt{Integer}.
\[
\texttt{hcf} :: \texttt{Integer} \rightarrow \texttt{Integer} \rightarrow \texttt{Integer}
\texttt{hcf \ x \ 0} = \texttt{x}
\texttt{hcf \ x \ y} = \texttt{hcf \ y \ (rem \ x \ y)}
\]

9. Write a Haskell function \texttt{relprime} :: \texttt{Integer} \rightarrow \texttt{Integer} \rightarrow \texttt{Bool}, that is true if it’s arguments are relatively prime, but false otherwise.
\[
\texttt{relprime} :: \texttt{Integer} \rightarrow \texttt{Integer} \rightarrow \texttt{Bool}
\texttt{relprime \ x \ y} = \texttt{hcf \ x \ y == 1}
\]

10. The Euler totient (or phi) function \texttt{euler \ n} is a count of how many numbers less than \(n\) are relatively prime to \(n\), e.g. \texttt{euler 6} is 2 because 1 and 5 are relatively prime to 6. Write a Haskell function \texttt{euler} :: \texttt{Integer} \rightarrow \texttt{Int} that calculates the totient of it’s argument.
euler :: Integer -> Int
euler n = length (filter (relprime n) [1 .. n-1])

11. Write a Haskell function `sumTotient :: Integer -> Integer -> Int`, so that `sumTotient lower upper` calculates the sum of the totients between lower and upper.

\[
\text{sumTotient :: Integer -> Integer -> Int}
\]
\[
\text{sumTotient lower upper = sum (map euler [lower, lower+1 .. upper])}
\]

12. (a) Show that Java is a referentially opaque notation.

Java has an assignment statement, e.g. \( x = x + 1 \), and hence the value of the expression \( x \) is not “the same wherever it occurs.”

(b) Is the French language referentially transparent or opaque? Justify your answer.

It is opaque because it is possible for a sentence to refer to itself.

(c) It is relatively easy to reason about referentially transparent notations like Haskell. Prove that in Haskell, if \( f \ x = x + 1 \), then \( 2 * (f \ x) = (f \ x) + (f \ x) \)

\[
\begin{array}{ll}
\text{L.H.S} & \text{R.H.S.} \\
2 * (f \ x) & (f \ x) + (f \ x) \\
=> 2 * (x + 1) & => (x + 1) + (x + 1) \\
=> 2x + 2 & => 2x + 2 \\
=> \text{R.H.S.} &
\end{array}
\]